CSE343: Machine Learning Assignment-1

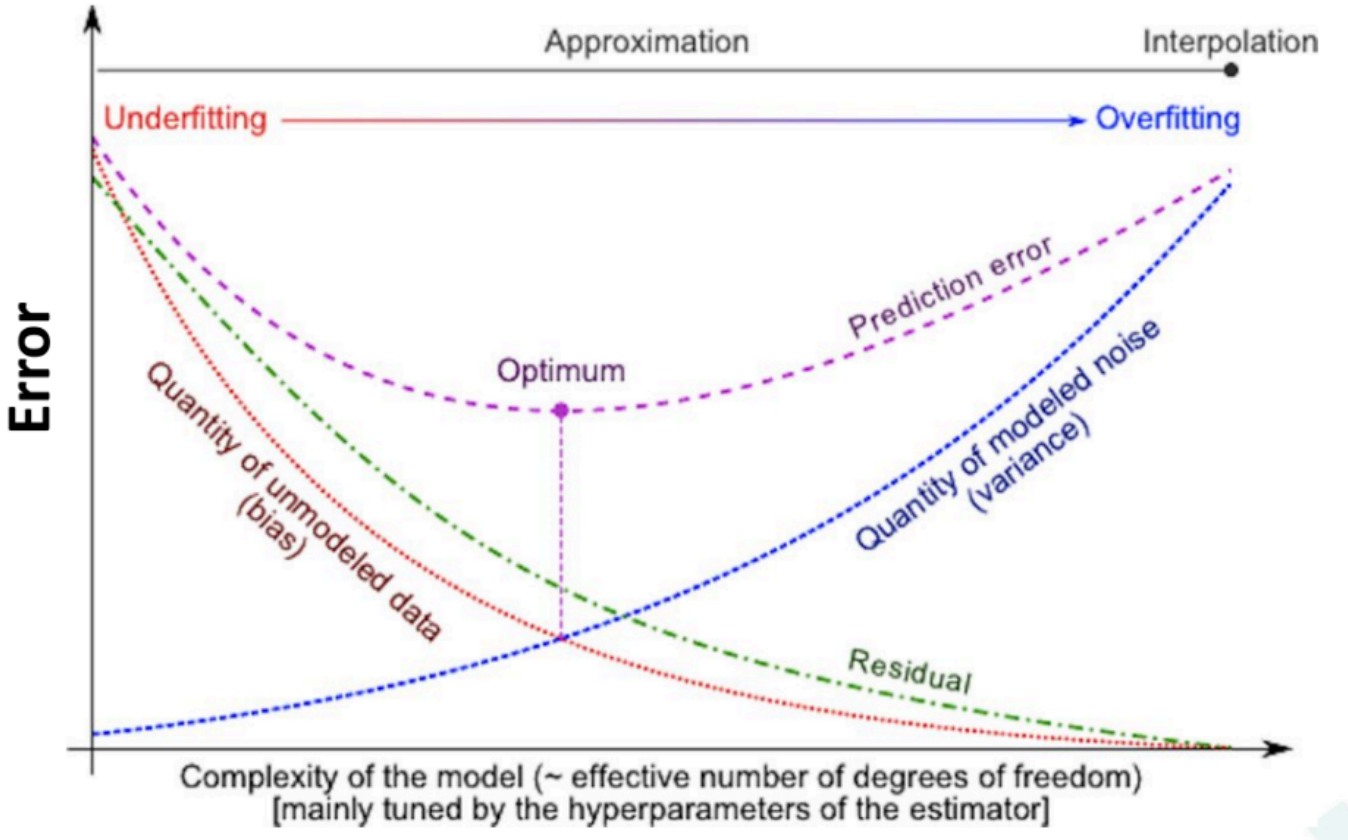
Ritika Thakur | 2022408

Section A: Theoretical

Question 1: You are developing a machine-learning model for a prediction task. As you increase the complexity of your model, for example, by adding more features or by including higher-order polynomial terms in a regression model, what is most likely to occur? Explain in terms of bias and variance with suitable graphs as applicable.

Ans: As we increase the complexity of our model by adding more features or by including higher-order polynomial terms in a regression model, the model will most likely overfit the data. Overfitting occurs when the model learns the training data too well, including the noise in the data, and fails to generalize to new, unseen data. This is because the model is too complex and has too many parameters, which allows it to fit the training data very closely but makes it less likely to generalize to new data. This overfitting will result in:

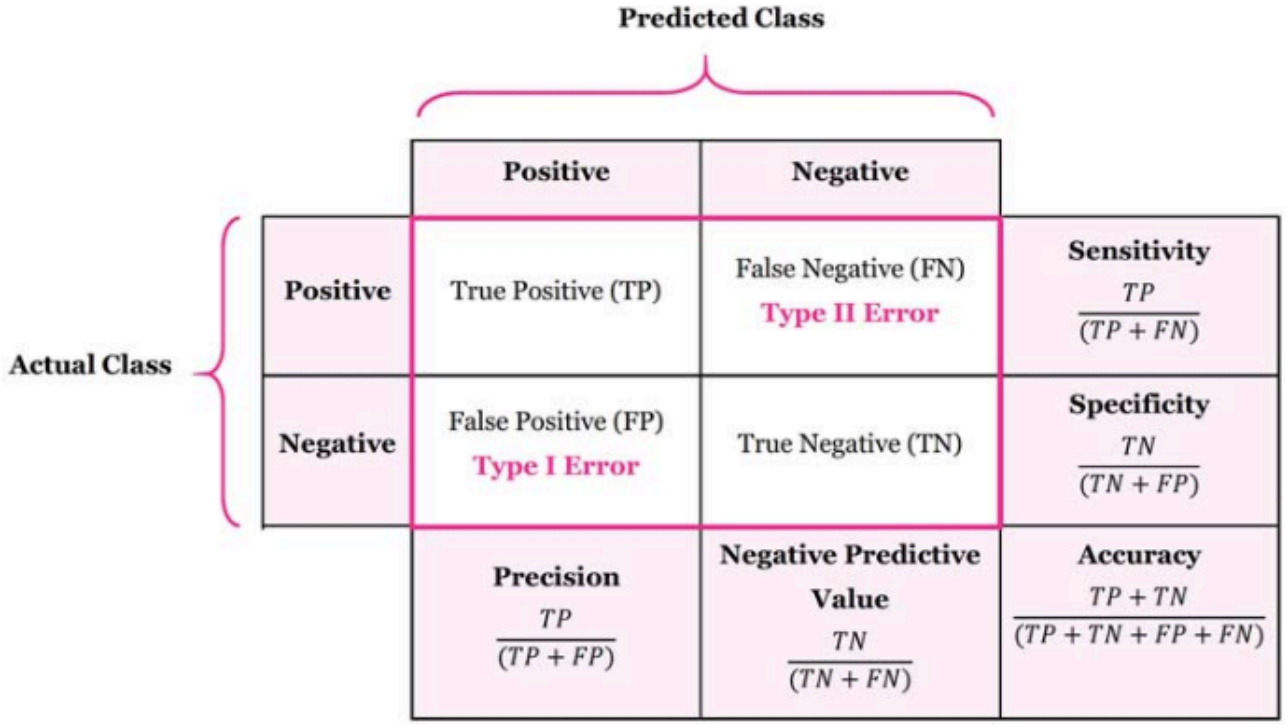
1. **Lower bias**: The model fits the training data very closely, thus reducing training error loweing the bias.
2. **Higher variance**: The model is too complex and fits the noise in the training data, which makes it less likely to generalize to new data, increasing the variance. Variance refers to the error due to the model's sensitivity to the training data.



Question 2: You're working at a tech company that has developed an advanced email filtering system to ensure users' inboxes are free from spam while safeguarding legitimate messages. After the model has been trained, you are tasked with evaluating its performance on a validation dataset containing a

mix of spam and legitimate emails. The results show that the model successfully identified 200 spam emails. However, 50 spam emails managed to slip through, being incorrectly classified as legitimate. Meanwhile, the system correctly recognised most of the legitimate emails, with 730 reaching the users' inboxes as intended. Unfortunately, the filter mistakenly flagged 20 legitimate emails as spam, wrongly diverting them to the spam folder. You are asked to assess the model by calculating an average of its overall classification performance across the different categories of emails.

Ans: We will use the below metric to evaluate the model's performance:



1. True Positive (TP): The number of spam emails correctly identified by the model = 200
2. False Negative (FN): The number of spam emails incorrectly classified as legitimate = 50
3. True Negative (TN): The number of legitimate emails correctly identified by the model = 730
4. False Positive (FP): The number of legitimate emails incorrectly classified as spam = 20

Accuracy = (TP + TN) / (TP + TN + FP + FN) = (200 + 730) / (200 + 730 + 20 + 50) = 930 / 1000 = 0.93 or 93%

Precision = TP / (TP + FP) = 200 / (200 + 20) = 200 / 220 = 0.909090... or 91%

Recall = TP / (TP + FN) = 200 / (200 + 50) = 200 / 250 = 0.8 or 80%

Specificity = TN / (TN + FP) = 730 / (730 + 20) = 730 / 750 = 0.9733 or 97.33%

Negative Predictive Value = TN / (TN + FN) = 730 / (730 + 50) = 730 / 780 = 0.935 or 93.5%

F1 Score = 2 * (Precision * Recall) / (Precision + Recall) = 2 * (0.909090... * 0.8) / (0.909090... + 0.8) = 1.454545... / 1.709090... = 0.85 or 85%

From the above calculations, we can see that the model has an accuracy of 93%, a precision of 91%, a recall of 80%, a specificity of 97.33%, a negative predictive value of 93.5%, and an F1 score of 85%.
These metrics indicate that the model performs well in identifying legitimate emails but has a lower recall for spam emails, meaning that it misses some spam emails.

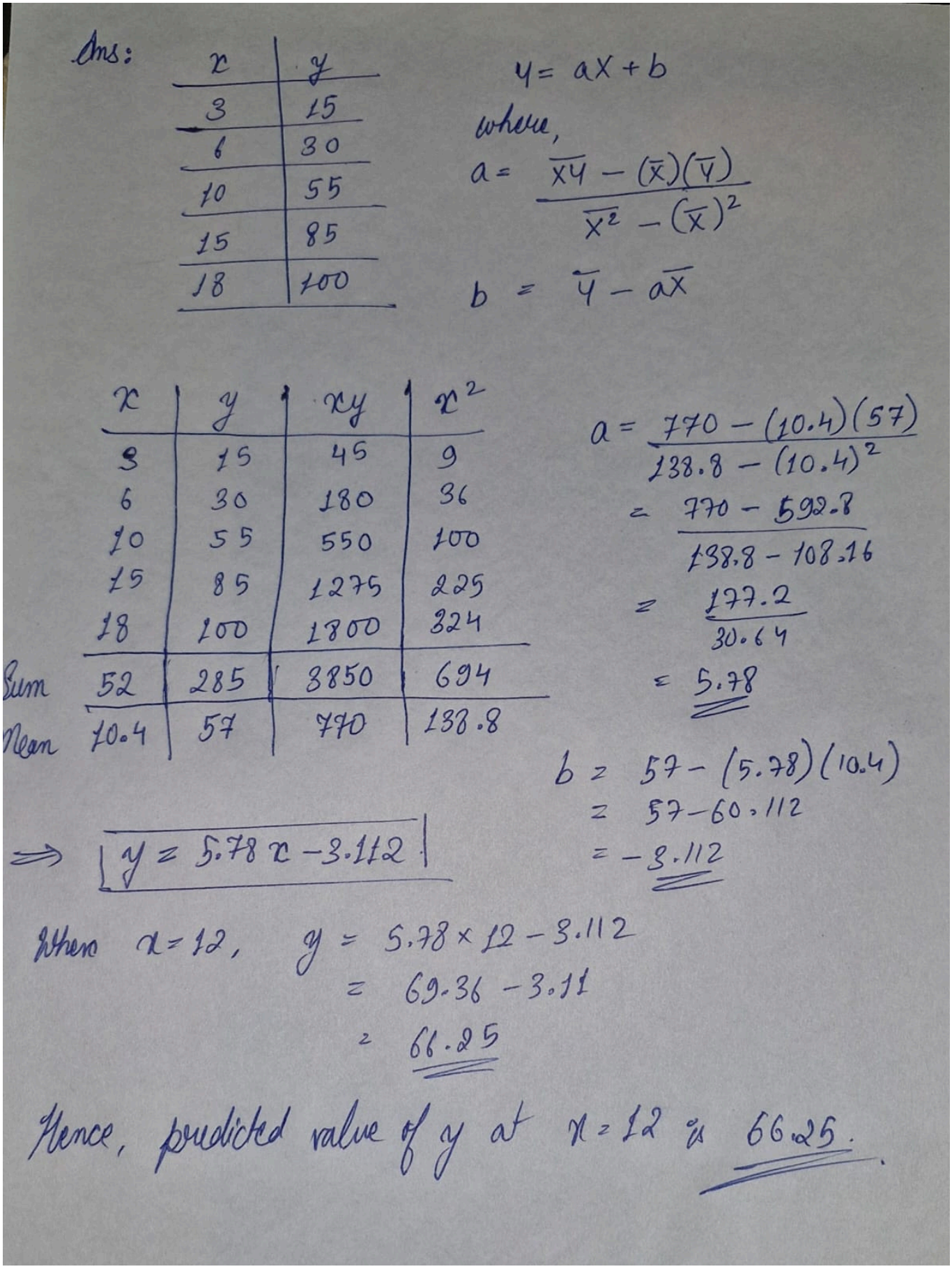
The F1 score, which is the harmonic mean of precision and recall, is 85%, indicating that the model has a good balance between precision and recall.

Question 3: Consider the following data where y(units) is related to x(units) over a period of time: Find the equation of the regression line and, using the regression equation obtained, predict the value of y when x = 12.

x	y
3	15
6	30
10	55
15	85
18	100

Table 1: Table of x and y values

Ans:



Question 4: Given a training dataset with features X and labels Y , let f(X) be the prediction of a model f and L(f(X), Y) be the loss function. Suppose you have two models, f1 and f2, and the empirical risk for f1 is lower than that for f2. Provide a toy example where model f1 has a lower empirical risk on the training set but may not necessarily generalize better than model f2.

Ans: If f1 has a lower empirical risk than f2 on the training set but does not necessarily generalise on the testing set then we are talking about the case of overfitting. Overfitting occurs when the model learns the training data too well, including the noise in the data, and fails to generalize to new, unseen data. This is because the model is too complex and has too many parameters, which allows it to fit the training data very closely but makes it less likely to generalize to new data.

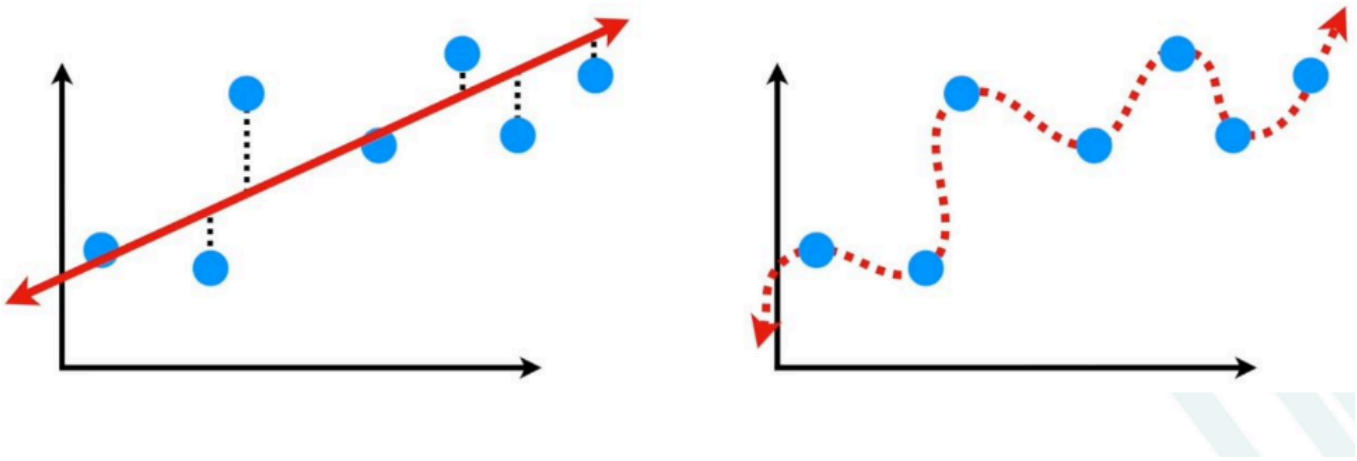


Image on left is the model f2: a linear model which shows a higher bias on the training data as compared to the image on the right in which the model f1: a polynomial model which shows a lower bias on the training data.

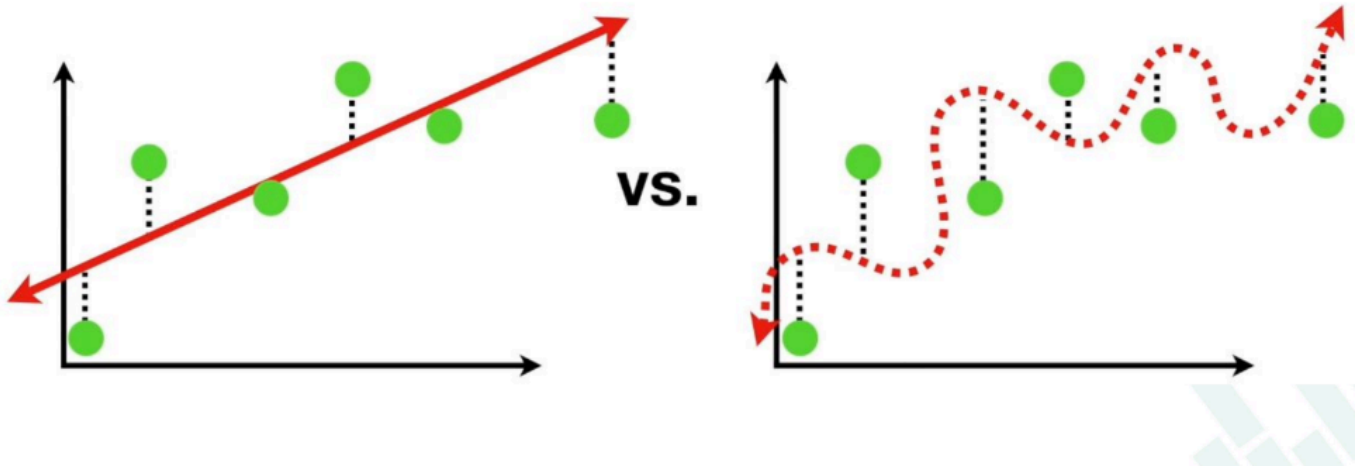


Image on left is the model f2: a linear model which shows a lower variance on the testing data as compared to the image on the right in which the model f1: a polynomial model which shows a higher variance on the testing data.

Let us take a toy example to illustrate this scenario:

Ans:

$f1 = \frac{x^2}{6}$

$f2 = 1x + 1$

Training Set:

X	Y	f1(x)	f2(x)
1	0.1	0.167	0
2	0.6	0.67	1
3	1.5	1.5	2
4	2.5	2.67	3
5	4	4.16	4

Absolute loss for f1(x)

= 0.067 + 0.07 + 0 + 0.17 + 0.16

= 0.467

Absolute loss for f2(x)

= 0.1 + 0.4 + 0.5 + 0.5 + 0

= 1.5

⇒ Absolute loss for f1(x) < Absolute loss for f2(x)

⇒ $f1 = x^2/6$ performs better on the training data

Testing Set:

X	Y	f1(x)	f2(x)
6	5.5	6	5
7	6.6	8.167	6
8	7.5	10.67	7

Absolute loss for f1(x)

= 0 + 1.567 + 3.17

= 4.737

Absolute loss for f2(x)

= 0.5 + 0.6 + 0.5

= 1.6

⇒ Absolute loss for f1(x) > Absolute loss for f2(x)

⇒ $f1 = x^2/6$ overfits and fails to generalize to testing data compared to $f2 = 1x - 1$.

In [349…
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Loading and Pre-processing the dataset

In [350…
```
data_frame = pd.read_csv('Heart Disease.csv')
# data_frame = data_frame.dropna()
data_frame
```

Out[350…

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevale
0	1	39	4.0	0	0.0	0.0	0	
1	0	46	2.0	0	0.0	0.0	0	
2	1	48	1.0	1	20.0	0.0	0	
3	0	61	3.0	1	30.0	0.0	0	
4	0	46	3.0	1	23.0	0.0	0	
…	…	…	…	…	…	…	…	
4233	1	50	1.0	1	1.0	0.0	0	
4234	1	51	3.0	1	43.0	0.0	0	
4235	0	48	2.0	1	20.0	NaN	0	
4236	0	44	1.0	1	15.0	0.0	0	
4237	0	52	2.0	0	0.0	0.0	0	

4238 rows × 16 columns

Dataset has 15 features and 1 target variable 'HeartDisease'.

In [351…
```
# preprocessing data by fillna mode for categorical data and mean for numerical dat
data_frame = data_frame.fillna({'male' : data_frame['male'].mode()[0],
                                'age' : data_frame['age'].mean(),
                                'education' : data_frame['education'].mode()[0],
                                'currentSmoker' : data_frame['currentSmoker'].mode()
                                'cigsPerDay' : data_frame['cigsPerDay'].mean(),
                                'BPMeds' : data_frame['BPMeds'].mode()[0],
                                'prevalentStroke' : data_frame['prevalentStroke'].mo
                                'prevalentHyp' : data_frame['prevalentHyp'].mode()[0
                                'diabetes' : data_frame['diabetes'].mode()[0],
                                'totChol' : data_frame['totChol'].mean(),
                                'sysBP' : data_frame['sysBP'].mean(),
                                'diaBP' : data_frame['diaBP'].mean(),
                                'BMI' : data_frame['BMI'].mean(),
```

```
                                    'heartRate' : data_frame['heartRate'].mean(),
                                    'glucose' : data_frame['glucose'].mean()})
```

Using fillna to fill the missing values of a categorical data with its mode and numerical data with its mean.

In [352…
```
print(data_frame.shape)
total_rows = data_frame.shape[0]

# train-validation-test split
data_frame = data_frame.sample(frac=1)
data_frame = data_frame.reset_index(drop=True)

train_df = data_frame[0 : int(total_rows * 0.7)]
val_df = data_frame[train_df.shape[0] : train_df.shape[0] + int(total_rows * 0.15)]
test_df = data_frame[train_df.shape[0] + val_df.shape[0] : ]

print(train_df.shape, val_df.shape, test_df.shape)
```

```
(4238, 16)
(2966, 16) (635, 16) (637, 16)
```

Splitting the dataset into 70% training 15% validation and 15% testing data.

In [353…
```
train_df
```

Out[353…

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevale
0	1	53	1.0	1	30.0	0.0	0	
1	0	39	3.0	0	0.0	0.0	0	
2	0	42	3.0	1	15.0	0.0	0	
3	0	61	2.0	0	0.0	0.0	0	
4	0	43	2.0	0	0.0	0.0	0	
...	...	...	...	...	...	...	...	
2961	1	64	1.0	1	20.0	0.0	0	
2962	1	51	1.0	0	0.0	0.0	0	
2963	1	38	2.0	1	20.0	0.0	0	
2964	0	36	2.0	0	0.0	0.0	0	
2965	1	44	1.0	0	0.0	0.0	0	

2966 rows × 16 columns

In [354…
```
val_df
```

Out[354...

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevale
2966	1	59	3.0	1	20.0	0.0	0	
2967	0	65	1.0	0	0.0	0.0	0	
2968	0	50	1.0	1	20.0	0.0	0	
2969	0	41	2.0	0	0.0	0.0	0	
2970	1	39	1.0	1	20.0	0.0	0	
...	...	...	...	...	...	...	...	
3596	0	46	3.0	1	20.0	0.0	0	
3597	0	56	1.0	1	9.0	0.0	0	
3598	0	58	2.0	1	3.0	0.0	0	
3599	0	45	3.0	1	15.0	0.0	0	
3600	0	45	3.0	1	20.0	0.0	0	

635 rows × 16 columns

In [355...
```
test_df
```

Out[355...

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	prevale
3601	1	40	2.0	1	30.0	0.0	0	
3602	0	61	2.0	0	0.0	1.0	0	
3603	0	54	1.0	1	20.0	0.0	0	
3604	1	68	1.0	0	0.0	0.0	0	
3605	1	58	1.0	1	20.0	0.0	0	
...	...	...	...	...	...	...	...	
4233	0	65	2.0	0	0.0	0.0	0	
4234	1	38	1.0	1	15.0	0.0	0	
4235	0	59	1.0	0	0.0	0.0	0	
4236	0	57	2.0	0	0.0	0.0	0	
4237	0	57	1.0	0	0.0	0.0	0	

637 rows × 16 columns

Splitting data into features and labels

In [356…
```
Y_train = train_df['HeartDisease']
X_train = train_df.drop(columns = ['HeartDisease'])

Y_val = val_df['HeartDisease']
X_val = val_df.drop(columns = ['HeartDisease'])

Y_test = test_df['HeartDisease']
X_test = test_df.drop(columns = ['HeartDisease'])
```

Helper functions: Sigmoid, Cross Entropy loss, Accuracy

In [357…
```
# creating helper functions for gradient descent

def initialize_weights(n):
    w = np.ones(n)
    b = 0
    return w, b

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def cross_entropy_loss(y, y_hat):
    m = y.shape[0]
    epsilon = 1e-6
    return -1/m * np.sum(y * np.log(y_hat + epsilon) + (1 - y) * np.log(1 - y_hat +

def accuracy(y, y_hat):
    y_hat = np.where(y_hat >= 0.5, 1, 0)
    return np.mean(y_hat == y)
```

(a) Gradient Descent on Unscaled Data

Function for gradient descent

In [358…
```
def gradient_descent(X_train, Y_train, X_val, Y_val, X_test, Y_test, learning_rate,
    W, b = initialize_weights(X_train.shape[1])

    train_loss = []
    val_loss = []
    test_loss = []

    train_accuracies = []
    val_accuracies = []
    test_accuracies = []

    Weights = []
    Biases = []

    m = X_train.shape[0]

    for epoch in range(epochs):
```

```
        z = np.dot(X_train, W) + b
        y_hat = sigmoid(z)
        loss = cross_entropy_loss(Y_train, y_hat)

        dW = 1/m * np.dot(X_train.T, (y_hat - Y_train))
        db = 1/m * np.sum(y_hat - Y_train)

        W -= learning_rate * dW
        b -= learning_rate * db

        train_loss.append(loss)
        val_loss.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_loss.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test, W) + b))

        train_accuracies.append(accuracy(Y_train, sigmoid(np.dot(X_train, W) + b)))
        val_accuracies.append(accuracy(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_accuracies.append(accuracy(Y_test, sigmoid(np.dot(X_test, W) + b)))

        Weights.append(W)
        Biases.append(b)

    return W, b, train_loss, val_loss, test_loss, train_accuracies, val_accuracies,
```

In [359…

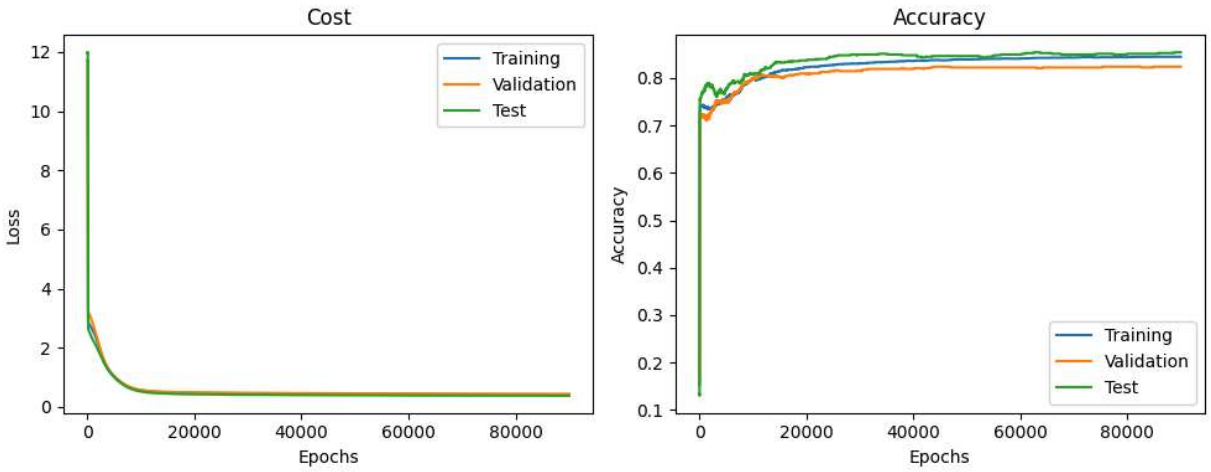
```
# batch gradient descent on unscaled data

epochs = 90000
learning_rate = 0.0001

W, b, train_loss, val_loss, test_loss, train_accuracy, val_accuracy, test_accuracy,
```

Plotting the loss and accuracy

In [360…

```
# plotting loss and accuracy
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='train loss')
plt.plot(val_loss, label='val loss')
plt.plot(test_loss, label='test loss')
plt.title('Cost')
plt.legend(['Training', 'Validation', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(train_accuracy, label='train accuracy')
plt.plot(val_accuracy, label='val accuracy')
plt.plot(test_accuracy, label='test accuracy')
plt.title('Accuracy')
plt.legend(['Training', 'Validation', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.tight_layout()
```

```
plt.show()

# printing the costs and accuracies
print("Epochs: ", epochs)
print("Learning Rate: ", learning_rate)

print("Training Loss: ", train_loss[-1])
print("Training Accuracy: ", train_accuracy[-1])
print("Validation Loss: ", val_loss[-1])
print("Validation Accuracy: ", val_accuracy[-1])
print("Test Loss: ", test_loss[-1])
print("Test Accuracy: ", test_accuracy[-1])

print("Weights: ", W)
print("Bias: ", b)
```




```
Epochs:  90000
Learning Rate:  0.0001
Training Loss:  0.4181924534665948
Training Accuracy:  0.844571813890762
Validation Loss:  0.4406239380431604
Validation Accuracy:  0.8236220472440945
Test Loss:  0.3820356875121941
Test Accuracy:  0.8540031397174255
Weights:  [ 0.85907561  0.02666912  0.1776508   0.84745543 -0.01809632  0.99880894
  1.00517791  1.03235256  1.01450175 -0.00232353  0.01237655 -0.03171568
 -0.05263322 -0.02246008  0.00111474]
Bias:  -0.08210944596265236
```

For 90000 iterations and learning rate 0.0001 we see our model starts converging after around 20000 iterations for unscaled data. The loss also stabilises to around 0.4 and accuracy to around 0.84 after these iterations. However the search path looks ineffecient as there is no stable increase in accuracy.

(b) Gradient Descent on Scaled Data

Min-Max Scaling

data = (data - data.min()) / (data.max() - data.min())

In [361…
```
# min-max scaling
X_train_scaled = (X_train - X_train.min()) / (X_train.max() - X_train.min())
X_val_scaled = (X_val - X_val.min()) / (X_val.max() - X_val.min())
X_test_scaled = (X_test - X_test.min()) / (X_test.max() - X_test.min())

W_scaled, b_scaled = initialize_weights(X_train_scaled.shape[1])
```

In [362…
```
X_train_scaled
```

Out[362…

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	pr
0	1.0	0.552632	0.000000	1.0	0.428571	0.0	0.0	
1	0.0	0.184211	0.666667	0.0	0.000000	0.0	0.0	
2	0.0	0.263158	0.666667	1.0	0.214286	0.0	0.0	
3	0.0	0.763158	0.333333	0.0	0.000000	0.0	0.0	
4	0.0	0.289474	0.333333	0.0	0.000000	0.0	0.0	
…	…	…	…	…	…	…	…	
2961	1.0	0.842105	0.000000	1.0	0.285714	0.0	0.0	
2962	1.0	0.500000	0.000000	0.0	0.000000	0.0	0.0	
2963	1.0	0.157895	0.333333	1.0	0.285714	0.0	0.0	
2964	0.0	0.105263	0.333333	0.0	0.000000	0.0	0.0	
2965	1.0	0.315789	0.000000	0.0	0.000000	0.0	0.0	

2966 rows × 15 columns

In [363…
```
X_val_scaled
```

Out[363...

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	pr
2966	1.0	0.702703	0.666667	1.0	0.333333	0.0	0.0	
2967	0.0	0.864865	0.000000	0.0	0.000000	0.0	0.0	
2968	0.0	0.459459	0.000000	1.0	0.333333	0.0	0.0	
2969	0.0	0.216216	0.333333	0.0	0.000000	0.0	0.0	
2970	1.0	0.162162	0.000000	1.0	0.333333	0.0	0.0	
...	...	...	...	...	...	...	...	
3596	0.0	0.351351	0.666667	1.0	0.333333	0.0	0.0	
3597	0.0	0.621622	0.000000	1.0	0.150000	0.0	0.0	
3598	0.0	0.675676	0.333333	1.0	0.050000	0.0	0.0	
3599	0.0	0.324324	0.666667	1.0	0.250000	0.0	0.0	
3600	0.0	0.324324	0.666667	1.0	0.333333	0.0	0.0	

635 rows × 15 columns

In [364...

```
X_test_scaled
```

Out[364...

	male	age	education	currentSmoker	cigsPerDay	BPMeds	prevalentStroke	pr
3601	1.0	0.171429	0.333333	1.0	0.500000	0.0	0.0	
3602	0.0	0.771429	0.333333	0.0	0.000000	1.0	0.0	
3603	0.0	0.571429	0.000000	1.0	0.333333	0.0	0.0	
3604	1.0	0.971429	0.000000	0.0	0.000000	0.0	0.0	
3605	1.0	0.685714	0.000000	1.0	0.333333	0.0	0.0	
...	...	...	...	...	...	...	...	
4233	0.0	0.885714	0.333333	0.0	0.000000	0.0	0.0	
4234	1.0	0.114286	0.000000	1.0	0.250000	0.0	0.0	
4235	0.0	0.714286	0.000000	0.0	0.000000	0.0	0.0	
4236	0.0	0.657143	0.333333	0.0	0.000000	0.0	0.0	
4237	0.0	0.657143	0.000000	0.0	0.000000	0.0	0.0	

637 rows × 15 columns

Running Gradient Descent on Scaled Data

In [365…

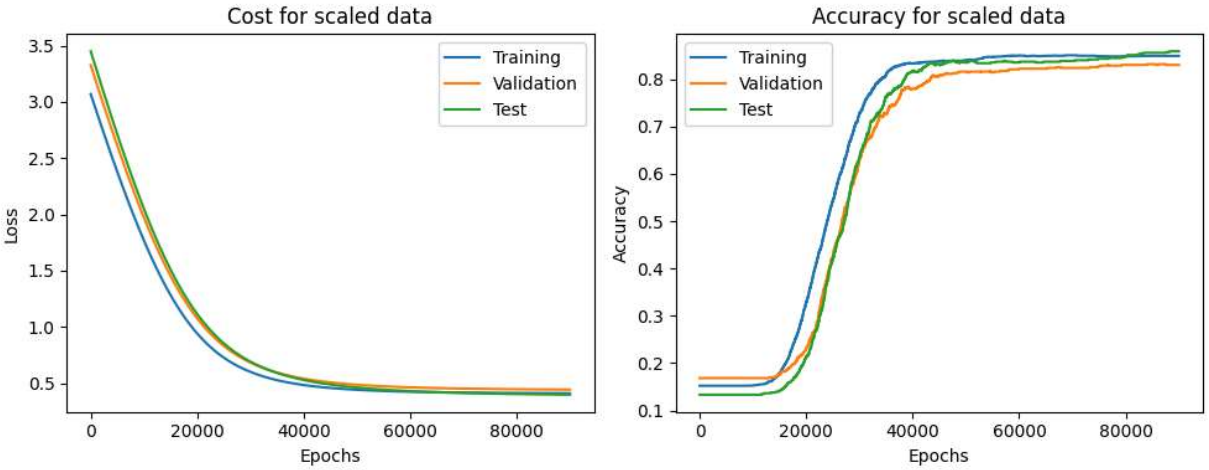
```
epochs = 90000
learning_rate = 0.0001

train_loss_scaled = []
val_loss_scaled = []
test_loss_scaled = []
Weights_scaled = []
Biases_scaled = []
train_accuracy_scaled = []
val_accuracy_scaled = []
test_accuracy_scaled = []

m = X_train_scaled.shape[0]

for epoch in range(epochs):
    z = np.dot(X_train_scaled, W_scaled) + b_scaled
    y_hat = sigmoid(z)
    loss = cross_entropy_loss(Y_train, y_hat)

    dW = 1 / m * np.dot(X_train_scaled.T, (y_hat - Y_train))
    db = 1 / m * np.sum(y_hat - Y_train)

    W_scaled -= learning_rate * dW
    b_scaled -= learning_rate * db

    train_loss_scaled.append(loss)
    val_loss_scaled.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val_scaled, W
    test_loss_scaled.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test_scaled

    Weights_scaled.append(W_scaled)
    Biases_scaled.append(b_scaled)

    train_accuracy_scaled.append(accuracy(Y_train, y_hat))
    val_accuracy_scaled.append(accuracy(Y_val, sigmoid(np.dot(X_val_scaled, W_scale
    test_accuracy_scaled.append(accuracy(Y_test, sigmoid(np.dot(X_test_scaled, W_sc
```

Plotting the loss and accuracy

In [366…

```
# plotting loss and accuracy
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss_scaled, label='train loss')
plt.plot(val_loss_scaled, label='val loss')
plt.plot(test_loss_scaled, label='test loss')
plt.title('Cost for scaled data')
plt.legend(['Training', 'Validation', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(train_accuracy_scaled, label='train accuracy')
plt.plot(val_accuracy_scaled, label='val accuracy')
plt.plot(test_accuracy_scaled, label='test accuracy')
plt.title('Accuracy for scaled data')
```

```
plt.legend(['Training', 'Validation', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()

# printing the costs and accuracies
print("Epochs: ", epochs)
print("Learning Rate: ", learning_rate)

print("Training Loss: ", train_loss_scaled[-1])
print("Training Accuracy: ", train_accuracy_scaled[-1])
print("Validation Loss: ", val_loss_scaled[-1])
print("Validation Accuracy: ", val_accuracy_scaled[-1])
print("Test Loss: ", test_loss_scaled[-1])
print("Test Accuracy: ", test_accuracy_scaled[-1])

print("Weights: ", W_scaled)
print("Bias: ", b_scaled)
```




```
Epochs:  90000
Learning Rate:  0.0001
Training Loss:  0.4093533746369167
Training Accuracy:  0.8489548213081591
Validation Loss:  0.4419667254702871
Validation Accuracy:  0.8299212598425196
Test Loss:  0.39672846623241065
Test Accuracy:  0.858712715855573
Weights:  [ 0.04814814  0.02337468  0.15881095 -0.16912908  0.70053126  0.92317605
  0.99328019  0.28630378  0.93980349  0.54514835  0.4968039   0.15771707
  0.41777361  0.21889135  0.73165811]
Bias:  -2.316239688197629
```

For 90000 iterations and learning rate 0.0001 we see our model starts converging between 20000 to 40000 iterations for scaled data. The loss also stabilises to around 0.4 and accuracy to around 0.84 after these iterations. The search path looks stable and we see a more stable convergence for scaled data when we compare it with unscaled data.

(c) Performance metrics

In [367…

```
def performance(Y, Y_pred_prob):
    # Convert probabilities to binary predictions using a 0.5 threshold for other m
    Y_pred = np.where(Y_pred_prob >= 0.5, 1, 0)

    # True Positives, True Negatives, False Positives, False Negatives
    TP = np.sum((Y == 1) & (Y_pred == 1))
    TN = np.sum((Y == 0) & (Y_pred == 0))
    FP = np.sum((Y == 0) & (Y_pred == 1))
    FN = np.sum((Y == 1) & (Y_pred == 0))

    print("True Positives: ", TP)
    print("True Negatives: ", TN)
    print("False Positives: ", FP)
    print("False Negatives: ", FN)

    # Precision
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    # Recall
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    # F1 Score
    f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0

    # Confusion Matrix
    confusion_matrix = np.array([[TN, FP], [FN, TP]])
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.imshow(confusion_matrix, cmap='YlOrRd', interpolation='nearest')
    plt.title('Confusion Matrix')
    plt.colorbar()
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.xticks([0, 1])
    plt.yticks([0, 1])

    # ROC AUC Curve
    thresholds = np.linspace(0, 1, 100)  # Create a range of thresholds between 0 a
    tpr_lst = []
    fpr_lst = []

    for threshold in thresholds:
        # Thresholding Y_pred_prob for ROC
        Y_pred_threshold = np.where(Y_pred_prob >= threshold, 1, 0)

        TP = np.sum((Y == 1) & (Y_pred_threshold == 1))
        TN = np.sum((Y == 0) & (Y_pred_threshold == 0))
        FP = np.sum((Y == 0) & (Y_pred_threshold == 1))
        FN = np.sum((Y == 1) & (Y_pred_threshold == 0))

        tpr = TP / (TP + FN) if (TP + FN) > 0 else 0  # True Positive Rate (Recall)
        fpr = FP / (FP + TN) if (FP + TN) > 0 else 0  # False Positive Rate
```

```
        tpr_lst.append(tpr)
        fpr_lst.append(fpr)

    tpr_lst = np.array(tpr_lst)
    fpr_lst = np.array(fpr_lst)

    # Plot ROC curve
    plt.subplot(1, 2, 2)
    plt.plot(fpr_lst, tpr_lst, color='darkorange', lw=2, label='ROC curve')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')  # Diagonal line
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(['ROC curve'])
    plt.tight_layout()
    plt.show()

    # AUC Calculation
    auc = abs(np.trapezoid(tpr_lst, fpr_lst))
    print("AUC: ", auc)


    # fpr = FP / (FP + TN)
    # tpr = TP / (TP + FN)

    # fpr_lst = []
    # tpr_lst = []

    # for i in range(len(Y_pred)):
    #     fpr_lst.append(FP / (FP + TN))
    #     tpr_lst.append(TP / (TP + FN))

    # plt.subplot(1, 2, 2)
    # plt.plot(fpr, tpr, 'ro')
    # plt.title('ROC Curve')
    # plt.xlabel('False Positive Rate')
    # plt.ylabel('True Positive Rate')

    # plt.tight_layout()
    # plt.show()

    # plt.figure()
    # plt.plot(fpr_lst, tpr_lst, color='darkorange', lw=2, label='ROC curve')
    # plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    # plt.xlim([0.0, 1.0])
    # plt.ylim([0.0, 1.05])
    # plt.xlabel('False Positive Rate')
    # plt.ylabel('True Positive Rate')
    # plt.title('Receiver Operating Characteristic')
    # plt.legend(loc="lower right")
    # plt.show()

    print("Precision: ", precision)
    print("Recall: ", recall)
```

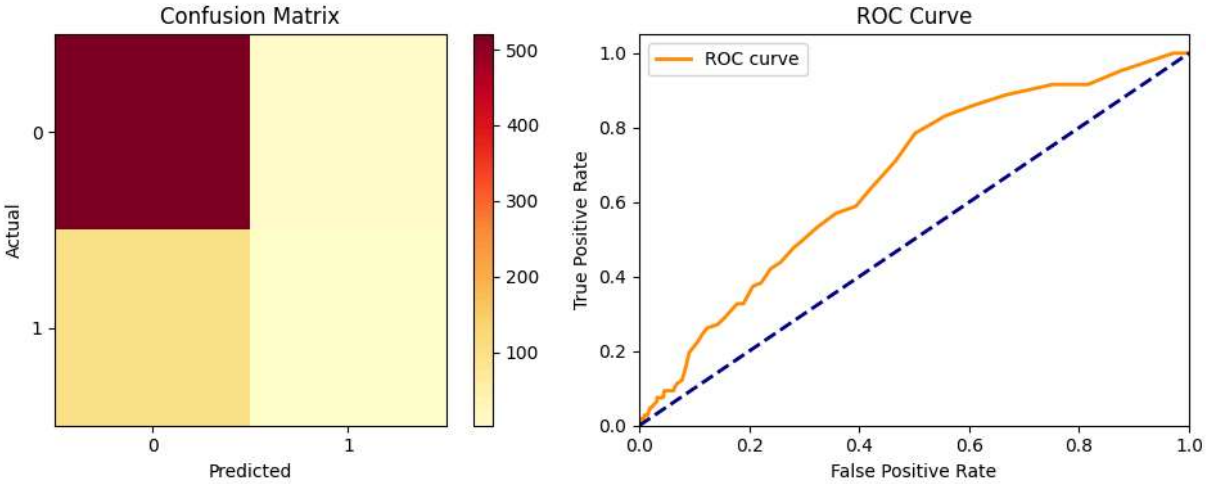
```
    print("F1 Score: ", f1)

    return precision, recall, f1

def performance_inv(Y, Y_pred_prob):
    # Convert probabilities to binary predictions using a 0.5 threshold
    Y_pred = np.where(Y_pred_prob >= 0.5, 1, 0)

    # True Positives, True Negatives, False Positives, False Negatives (inverted lo
    TP = np.sum((Y == 0) & (Y_pred == 0))  # Inverted: TP are the correctly predict
    TN = np.sum((Y == 1) & (Y_pred == 1))  # Inverted: TN are the correctly predict
    FP = np.sum((Y == 1) & (Y_pred == 0))  # Inverted: FP are the 1s incorrectly pr
    FN = np.sum((Y == 0) & (Y_pred == 1))  # Inverted: FN are the 0s incorrectly pr

    print("True Positives: ", TP)
    print("True Negatives: ", TN)
    print("False Positives: ", FP)
    print("False Negatives: ", FN)

    # Precision
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    # Recall
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    # F1 Score
    f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0

    # Confusion Matrix
    confusion_matrix = np.array([[TN, FP], [FN, TP]])  # Maintain the structure for
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.imshow(confusion_matrix, cmap='YlOrRd', interpolation='nearest')
    plt.title('Confusion Matrix')
    plt.colorbar()
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.xticks([0, 1])
    plt.yticks([0, 1])

    # ROC AUC Curve (Use probability scores for different thresholds)
    thresholds = np.linspace(0, 1, 100)
    tpr_lst = []
    fpr_lst = []

    for threshold in thresholds:
        # Thresholding Y_pred_prob for ROC
        Y_pred_threshold = np.where(Y_pred_prob >= threshold, 1, 0)

        TP = np.sum((Y == 0) & (Y_pred_threshold == 0))
        TN = np.sum((Y == 1) & (Y_pred_threshold == 1))
        FP = np.sum((Y == 1) & (Y_pred_threshold == 0))
        FN = np.sum((Y == 0) & (Y_pred_threshold == 1))

        tpr = TP / (TP + FN) if (TP + FN) > 0 else 0  # True Positive Rate (Recall)
        fpr = FP / (FP + TN) if (FP + TN) > 0 else 0  # False Positive Rate

        tpr_lst.append(tpr)
```

```
        fpr_lst.append(fpr)

    tpr_lst = np.array(tpr_lst)
    fpr_lst = np.array(fpr_lst)

    # Plot ROC curve
    plt.subplot(1, 2, 2)
    plt.plot(fpr_lst, tpr_lst, color='darkorange', lw=2, label='ROC curve')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')  # Diagonal line
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(['ROC curve'])
    plt.tight_layout()
    plt.show()

    # AUC Calculation
    auc = np.trapezoid(tpr_lst, fpr_lst)
    print("AUC: ", auc)

    print("Precision: ", precision)
    print("Recall: ", recall)
    print("F1 Score: ", f1)

    return precision, recall, f1
```

In [368…

```
performance(Y_val, sigmoid(np.dot(X_val, W) + b))
```

```
True Positives:  3
True Negatives:  520
False Positives:  8
False Negatives:  104
```




```
AUC:  0.6591794109317474
Precision:  0.2727272727272727
Recall:  0.028037383177570093
F1 Score:  0.05084745762711864
```

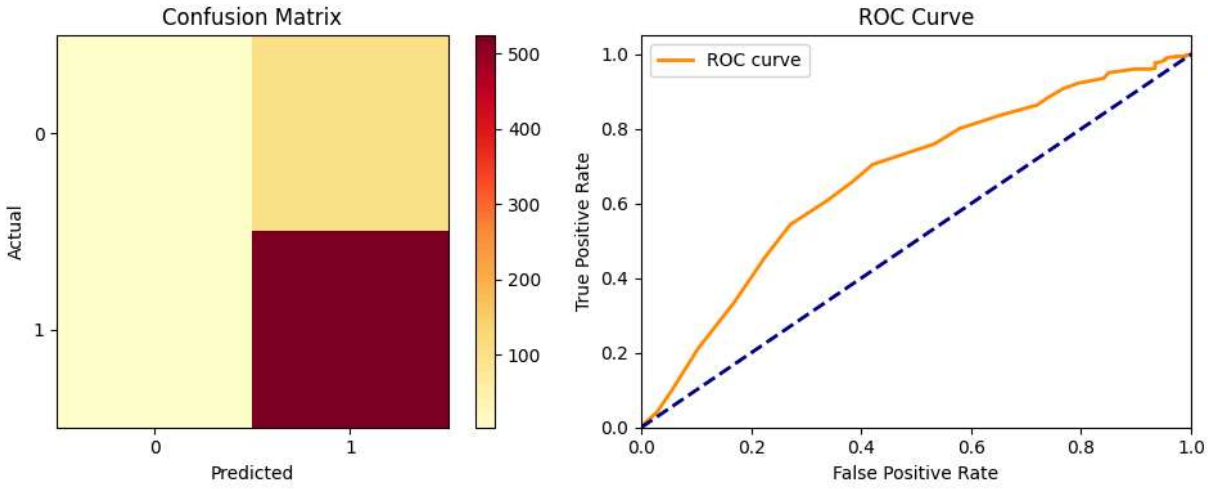
Out[368… (np.float64(0.2727272727272727),
 np.float64(0.028037383177570093),
 np.float64(0.05084745762711864))

When we take class 1 as positive and class 0 as negative, we see a large number of true negatives and false negatives and an auc score of around 0.65 which suggests that our model is better than random guessing but still has room for improvement. The the number of false and true negatives are due to our dataset being comprised of mostly class 0 values (Which we take as negative here).

The ROC curve is used to evaluate the performance of a binary classifier. It is a plot of the false positive rate (x-axis) versus the true positive rate (y-axis) for a number of different candidate threshold values between 0.0 and 1.0. The AUC is the area under the ROC curve. The AUC represents a model's ability to discriminate between positive and negative classes. An area of 1.0 represents a model that made all predictions perfectly. An area of 0.5 represents a model as good as random.

In [369…
```
performance_inv(Y_val, sigmoid(np.dot(X_val_scaled, W_scaled) + b_scaled))
```

```
True Positives:  524
True Negatives:  3
False Positives:  104
False Negatives:  4
```




```
AUC:  0.6656311951288587
Precision:  0.8343949044585988
Recall:  0.9924242424242424
F1 Score:  0.9065743944636678
```

Out[369… (np.float64(0.8343949044585988),
 np.float64(0.9924242424242424),
 np.float64(0.9065743944636678))

When we take class 0 as positive and class 1 as negative, we see a large number of true positives and false positives and an auc score of around 0.66 which suggests that our model is better than random guessing but still has room for improvement. The the number of false and true positives are due to our dataset being comprised of mostly class 0 values (Which we take as positive here).

(d) Stochastic Gradient Descent and Mini-batch Gradient Descent

Stochastic Gradient Descent

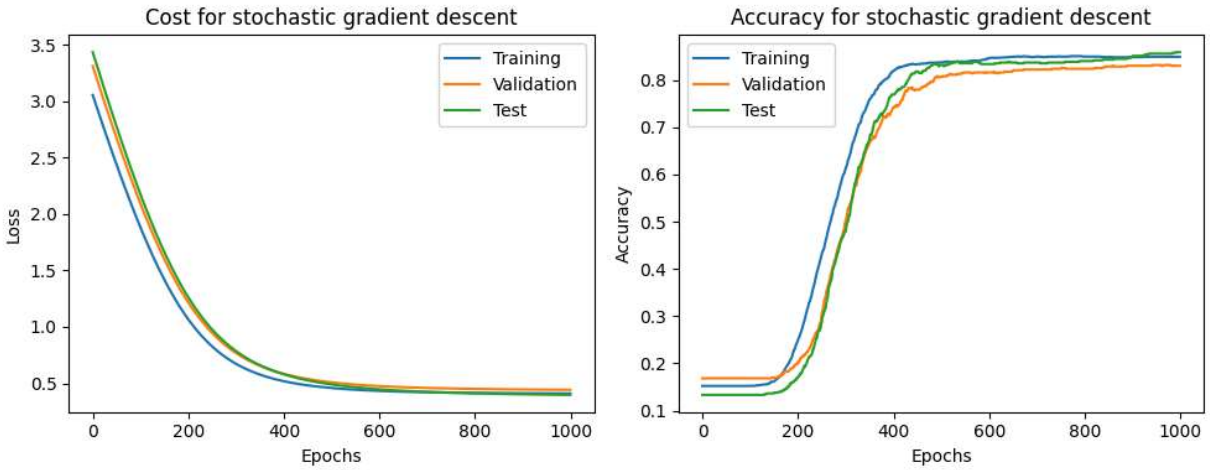
In [370…

```
# stochastic gradient descent

epochs = 1000
learning_rate = 0.000003

W_stoch, b_stoch = initialize_weights(X_train.shape[1])

train_loss_stoch = []
val_loss_stoch = []
test_loss_stoch = []
Weights_stoch = []
Biases_stoch = []
train_accuracy_stoch = []
val_accuracy_stoch = []
test_accuracy_stoch = []

m = X_train.shape[0]

for epoch in range(epochs):
    X_sample = X_train_scaled.sample(frac = 1)
    Y_sample = Y_train.loc[X_sample.index]

    y_hat = sigmoid(np.dot(X_sample, W_stoch) + b_stoch)

    dW = np.dot(X_sample.T, (y_hat - Y_sample))

    db = np.sum(y_hat - Y_sample)

    W_stoch -= learning_rate * dW
    b_stoch -= learning_rate * db

    train_loss_stoch.append(cross_entropy_loss(Y_train, sigmoid(np.dot(X_train_scal
    val_loss_stoch.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val_scaled, W_
    test_loss_stoch.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test_scaled,

    Weights_stoch.append(W_stoch)
    Biases_stoch.append(b_stoch)

    train_accuracy_stoch.append(accuracy(Y_train, sigmoid(np.dot(X_train_scaled, W_
    val_accuracy_stoch.append(accuracy(Y_val, sigmoid(np.dot(X_val_scaled, W_stoch)
    test_accuracy_stoch.append(accuracy(Y_test, sigmoid(np.dot(X_test_scaled, W_sto
```

Plotting the loss and accuracy

In [371…

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss_stoch, label='train loss')
plt.plot(val_loss_stoch, label='val loss')
plt.plot(test_loss_stoch, label='test loss')
plt.title('Cost for stochastic gradient descent')
plt.legend(['Training', 'Validation', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(train_accuracy_stoch, label='train accuracy')
plt.plot(val_accuracy_stoch, label='val accuracy')
plt.plot(test_accuracy_stoch, label='test accuracy')
plt.title('Accuracy for stochastic gradient descent')
plt.legend(['Training', 'Validation', 'Test'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()

# printing the costs and accuracies
print("Epochs: ", epochs)
print("Learning Rate: ", learning_rate)

print("Training Loss: ", train_loss_stoch[-1])
print("Training Accuracy: ", train_accuracy_stoch[-1])
print("Validation Loss: ", val_loss_stoch[-1])
print("Validation Accuracy: ", val_accuracy_stoch[-1])
print("Test Loss: ", test_loss_stoch[-1])
print("Test Accuracy: ", test_accuracy_stoch[-1])

print("Weights: ", W_stoch)
print("Bias: ", b_stoch)
```



```
Epochs:  1000
Learning Rate:  3e-06
Training Loss:  0.40949817213933654
Training Accuracy:  0.8489548213081591
Validation Loss:  0.4421910726738169
Validation Accuracy:  0.8299212598425196
Test Loss:  0.39714522935068974
Test Accuracy:  0.858712715855573
Weights:  [ 0.04797243  0.02332972  0.16051648 -0.16844575  0.70060682  0.9234154
  0.99325079  0.28653171  0.93994577  0.54561547  0.49708334  0.15848834
  0.41843575  0.21987038  0.73190456]
Bias:  -2.313581488661836
```

For stochastic gradient descent we do not notice much of a convergence and the accuracy plot is noisy. This is beacause for each iteration we select a random sample and update the weights. This leads to a noisy search path.

The accuracy remainaing consistent throughout our codes suggest a poor dataset due to poor distribution of classes.

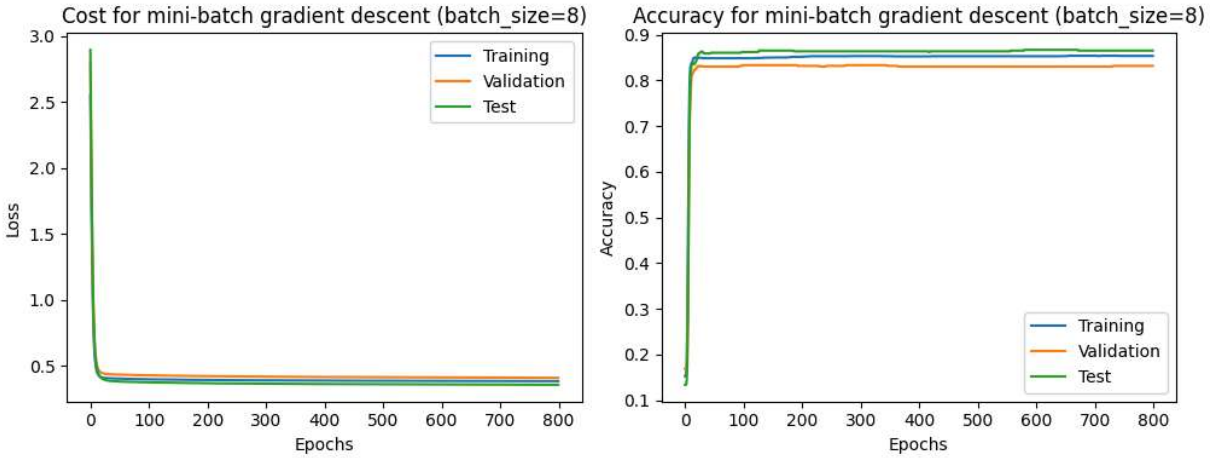
Mini-batch Gradient Descent

In [377…

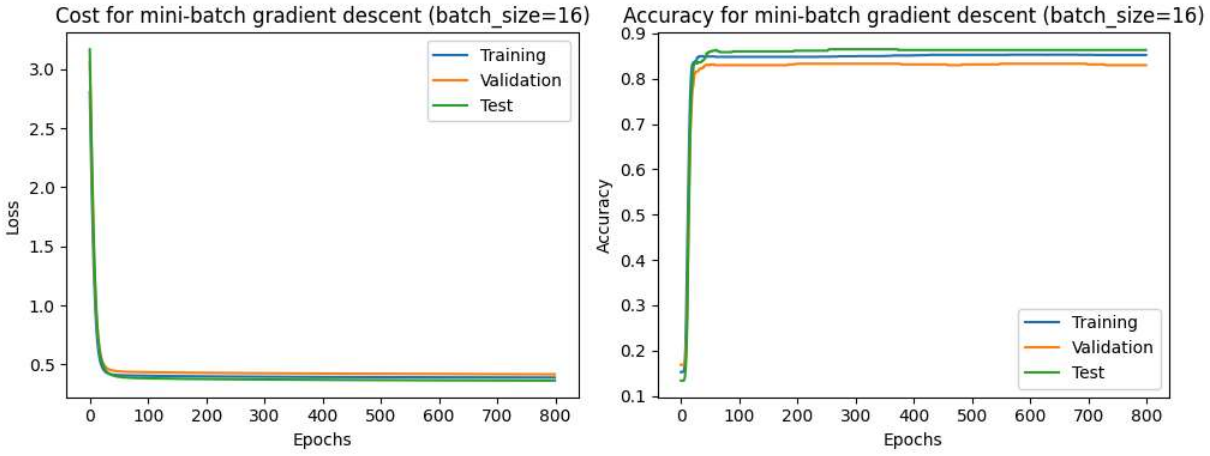
```
epochs = 800
learning_rate = 0.001
batch_sizes = [8, 16, 24, 32, 64]

for batch_size in batch_sizes:

    # Initialize weights
    W_mini, b_mini = initialize_weights(X_train.shape[1])

    train_loss_mini = []
    val_loss_mini = []
    test_loss_mini = []
    Weights_mini = []
    Biases_mini = []
    train_accuracy_mini = []
    val_accuracy_mini = []
    test_accuracy_mini = []

    m = X_train.shape[0]

    # Mini-batch gradient descent
    for epoch in range(epochs):
        # Shuffle training data at the start of each epoch
        shuffled_indices = np.random.permutation(m)
        X_train_shuffled = X_train_scaled.iloc[shuffled_indices].reset_index(drop=T
        Y_train_shuffled = Y_train.iloc[shuffled_indices].reset_index(drop=True)

        # Loop over batches
        for i in range(0, m, batch_size):
            X_sample = X_train_shuffled.iloc[i:i + batch_size]
            Y_sample = Y_train_shuffled.iloc[i:i + batch_size]
```

```
            # Forward pass
            y_hat = sigmoid(np.dot(X_sample, W_mini) + b_mini)

            # Compute gradients (averaged across the batch)
            dW = (1 / batch_size) * np.dot(X_sample.T, (y_hat - Y_sample))
            db = (1 / batch_size) * np.sum(y_hat - Y_sample)

            # Update weights
            W_mini -= learning_rate * dW
            b_mini -= learning_rate * db

        # Compute losses and accuracies at the end of the epoch
        train_loss = cross_entropy_loss(Y_train, sigmoid(np.dot(X_train_scaled, W_m
        val_loss = cross_entropy_loss(Y_val, sigmoid(np.dot(X_val_scaled, W_mini) +
        test_loss = cross_entropy_loss(Y_test, sigmoid(np.dot(X_test_scaled, W_mini

        train_loss_mini.append(train_loss)
        val_loss_mini.append(val_loss)
        test_loss_mini.append(test_loss)

        Weights_mini.append(W_mini)
        Biases_mini.append(b_mini)

        train_accuracy = accuracy(Y_train, sigmoid(np.dot(X_train_scaled, W_mini) +
        val_accuracy = accuracy(Y_val, sigmoid(np.dot(X_val_scaled, W_mini) + b_min
        test_accuracy = accuracy(Y_test, sigmoid(np.dot(X_test_scaled, W_mini) + b_

        train_accuracy_mini.append(train_accuracy)
        val_accuracy_mini.append(val_accuracy)
        test_accuracy_mini.append(test_accuracy)

    # Plotting the loss and accuracy (outside the epoch loop)
    plt.figure(figsize=(10, 4))

    # Loss plot
    plt.subplot(1, 2, 1)
    plt.plot(train_loss_mini, label='train loss')
    plt.plot(val_loss_mini, label='val loss')
    plt.plot(test_loss_mini, label='test loss')
    plt.title(f'Cost for mini-batch gradient descent (batch_size={batch_size})')
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    # Accuracy plot
    plt.subplot(1, 2, 2)
    plt.plot(train_accuracy_mini, label='train accuracy')
    plt.plot(val_accuracy_mini, label='val accuracy')
    plt.plot(test_accuracy_mini, label='test accuracy')
    plt.title(f'Accuracy for mini-batch gradient descent (batch_size={batch_size})'
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')

    plt.tight_layout()
```

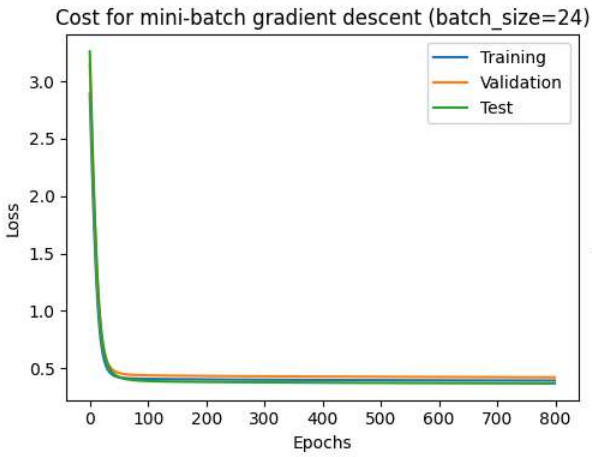
```
    plt.show()

    # Print final losses and accuracies for the current batch size
    print(f"Batch Size: {batch_size}")
    print("Final Training Loss: ", train_loss_mini[-1])
    print("Final Training Accuracy: ", train_accuracy_mini[-1])
    print("Final Validation Loss: ", val_loss_mini[-1])
    print("Final Validation Accuracy: ", val_accuracy_mini[-1])
    print("Final Test Loss: ", test_loss_mini[-1])
    print("Final Test Accuracy: ", test_accuracy_mini[-1])

    print("Final Weights: ", W_mini)
    print("Final Bias: ", b_mini)
```

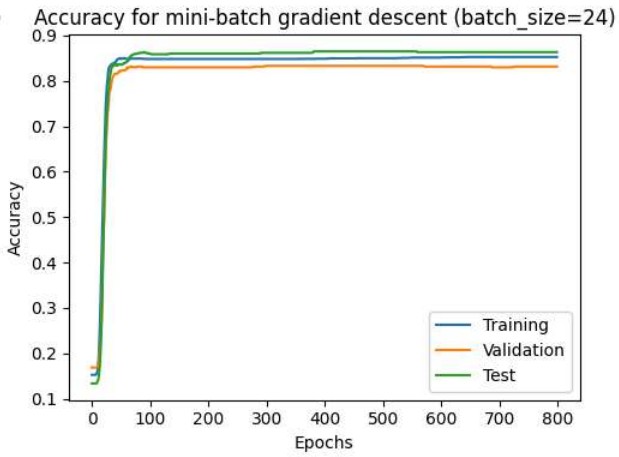



```
Batch Size: 8
Final Training Loss:  0.38517790867111024
Final Training Accuracy:  0.8533378287255563
Final Validation Loss:  0.4099052641417506
Final Validation Accuracy:  0.831496062992126
Final Test Loss:  0.3572096138100467
Final Test Accuracy:  0.8649921507064364
Final Weights:  [ 0.46024025  1.53954719 -0.30546564  0.0238545   0.95309363  0.6704
5648
  1.05909112  0.41656917  0.79715828  0.48296263  0.87372255  0.17869535
  0.14320649 -0.16397647  0.8159261 ]
Final Bias:  -3.4222734460141764
```

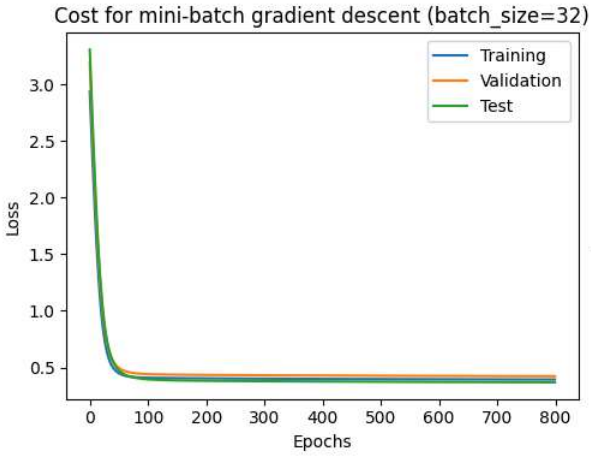


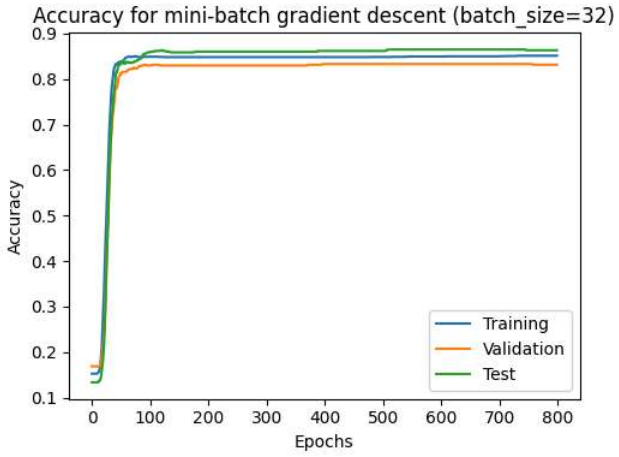
```
Batch Size: 16
Final Training Loss:  0.38887681097081633
Final Training Accuracy:  0.8523263654753878
Final Validation Loss:  0.4160305010939835
Final Validation Accuracy:  0.8299212598425196
Final Test Loss:  0.36256197331753576
Final Test Accuracy:  0.8634222919937206
Final Weights:  [ 0.44165713  0.97564804 -0.33743464 -0.02900371  0.8484818   0.7677
509
  1.03458107  0.47672932  0.864829    0.48502072  0.69120546  0.12277344
  0.23035973 -0.06463075  0.75185211]
Final Bias:  -3.0706807257177187
```



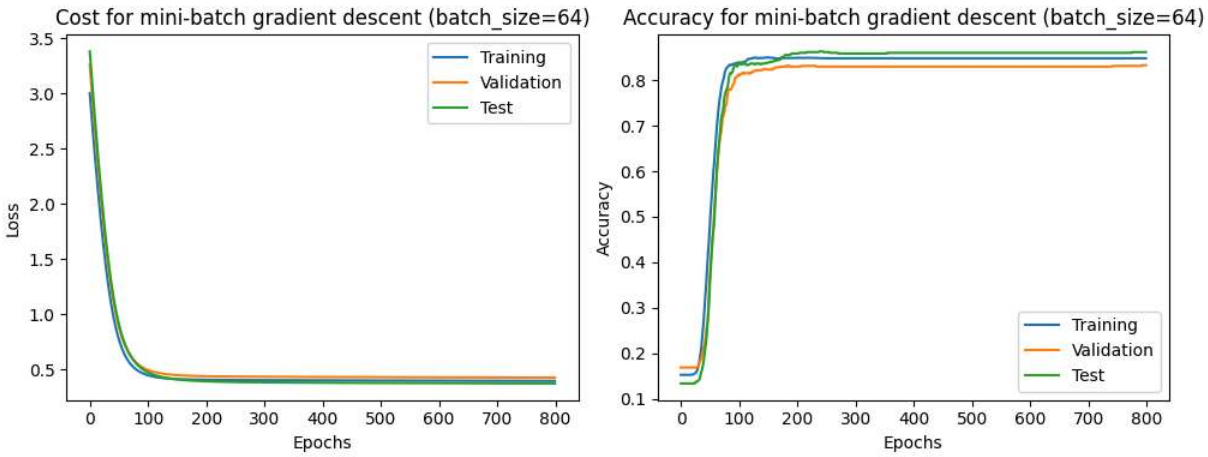



```
Batch Size: 24
Final Training Loss:  0.3914968223363446
Final Training Accuracy:  0.8526635198921105
Final Validation Loss:  0.4199677157174894
Final Validation Accuracy:  0.831496062992126
Final Test Loss:  0.366256337162806
Final Test Accuracy:  0.8634222919937206
Final Weights:  [ 0.40758004  0.68997184 -0.29644968 -0.06257679  0.80744585  0.8163
0892
  1.02374019  0.48135671  0.8920962   0.49017884  0.62335617  0.1155764
  0.2754767  -0.006169    0.73485272]
Final Bias:  -2.904633987201381
```





```
Batch Size: 32
Final Training Loss:  0.39347674604523214
Final Training Accuracy:  0.851652056641942
Final Validation Loss:  0.4226483799493182
Final Validation Accuracy:  0.831496062992126
Final Test Loss:  0.36898759833062217
Final Test Accuracy:  0.8634222919937206
Final Weights:  [ 0.36877843  0.52222161 -0.24500612 -0.08579681  0.78398785  0.8439
4058
  1.01723612  0.46801396  0.90605152  0.49529608  0.58741623  0.1156337
  0.30287729  0.03235326  0.72811788]
Final Bias:  -2.801788680353816
```




```
Batch Size: 64
Final Training Loss:  0.39860472810305647
Final Training Accuracy:  0.8482805124747134
Final Validation Loss:  0.42866034649048673
Final Validation Accuracy:  0.8330708661417323
Final Test Loss:  0.3757293718538606
Final Test Accuracy:  0.8618524332810047
Final Weights:  [ 0.24141451  0.23650417 -0.0908049  -0.13624323  0.74025978  0.8880
6234
  1.00513929  0.39839989  0.92582625  0.51010227  0.52985009  0.12217589
  0.35306761  0.11047669  0.722265  ]
Final Bias:  -2.600093890527015
```

For mini-batch gradient descent we see a more stable convergence and the accuracy plot is less noisy. This is beacause for each iteration we select a random batch of samples and update the weights. This leads to a better performance than stochastic gradient descent. However this method is computationally expensive as compared to stochastic gradient descent.

Mini batch gradient descent does reduce the complexity when compared to batch gradient descent but batch gradient descent is more stable and accurate.

(e) K-Fold Cross Validation

In [378…

```
folds = 5
epochs = 5000
```

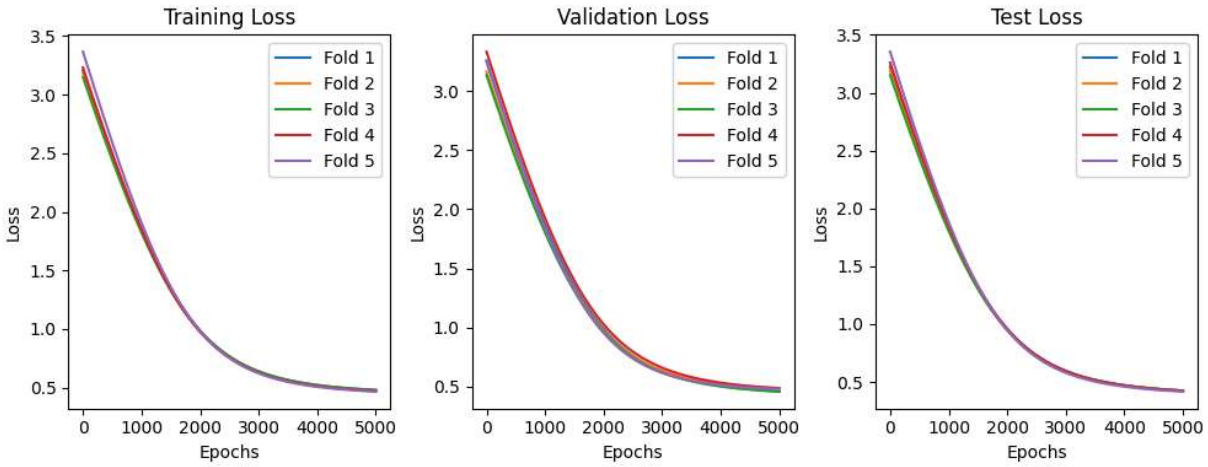
```
learning_rate = 0.001

# Combine train and validation sets
X_train_val = pd.concat([X_train, X_val])
Y_train_val = pd.concat([Y_train, Y_val])

# Shuffle the combined dataset
X_train_val = X_train_val.sample(frac=1, random_state=42).reset_index(drop=True)
Y_train_val = Y_train_val.loc[X_train_val.index].reset_index(drop=True)

# Calculate fold size
k = X_train_val.shape[0] // folds  # Integer division to avoid floating-point issue

fold_costs = []
fold_accuracies = []
fold_Ws = []
fold_bs = []

# Loop over each fold
for fold in range(folds):
    # Set aside the validation fold
    X_val_fold = X_train_val.iloc[fold * k : (fold + 1) * k].reset_index(drop=True)
    Y_val_fold = Y_train_val.iloc[fold * k : (fold + 1) * k].reset_index(drop=True)

    # Remaining as the training set
    X_train_fold = X_train_val.drop(X_train_val.index[fold * k : (fold + 1) * k]).r
    Y_train_fold = Y_train_val.drop(Y_train_val.index[fold * k : (fold + 1) * k]).r

    # Use training fold's min/max for scaling
    X_train_fold_min = X_train_fold.min()
    X_train_fold_max = X_train_fold.max()

    X_train_fold_scaled = (X_train_fold - X_train_fold_min) / (X_train_fold_max - X
    X_val_fold_scaled = (X_val_fold - X_train_fold_min) / (X_train_fold_max - X_tra

    # Scale the test set using min/max from the whole training set
    X_test_scaled = (X_test - X_train_fold_min) / (X_train_fold_max - X_train_fold_

    # Training using gradient descent
    W, b, train_costs, val_costs, test_costs, train_accuracies, val_accuracies, tes
        X_train_fold_scaled, Y_train_fold, X_val_fold_scaled, Y_val_fold, X_test_sc
    )

    # Store results for this fold
    fold_costs.append((train_costs, val_costs, test_costs))
    fold_accuracies.append((train_accuracies, val_accuracies, test_accuracies))
    fold_Ws.append(W)
    fold_bs.append(b)
```

Plotting the loss and accuracy

In [380…

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 3, 1)
for fold in fold_costs:
    plt.plot(fold[0])
```

```
plt.title('Training Loss')
plt.legend(['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5'])
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 3, 2)
for fold in fold_costs:
    plt.plot(fold[1])
plt.title('Validation Loss')
plt.legend(['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5'])
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.subplot(1, 3, 3)
for fold in fold_costs:
    plt.plot(fold[2])
plt.title('Test Loss')
plt.legend(['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5'])
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.tight_layout()
plt.show()

print("Average Training Loss: ", np.mean([fold[0][-1] for fold in fold_costs]))
print("Standard Deviation Training Loss: ", np.std([fold[0][-1] for fold in fold_co

print("Average Validation Loss: ", np.mean([fold[1][-1] for fold in fold_costs]))
print("Standard Deviation Validation Loss: ", np.std([fold[1][-1] for fold in fold_

print("Average Test Loss: ", np.mean([fold[2][-1] for fold in fold_costs]))
print("Standard Deviation Test Loss: ", np.std([fold[2][-1] for fold in fold_costs]
```

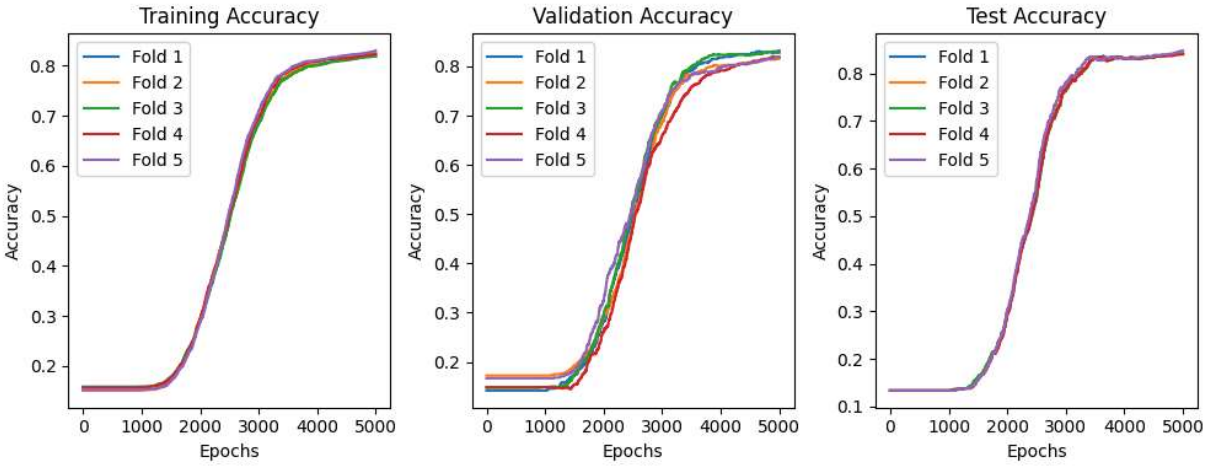



```
Average Training Loss:  0.4731127059065452
Standard Deviation Training Loss:  0.00514125393077108
Average Validation Loss:  0.47360918009246367
Standard Deviation Validation Loss:  0.012648189092234207
Average Test Loss:  0.4222149170465995
Standard Deviation Test Loss:  0.0028245293930817582
```

In [381…

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 3, 1)
```

```
for fold in fold_accuracies:
    plt.plot(fold[0])
plt.title('Training Accuracy')
plt.legend(['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1, 3, 2)
for fold in fold_accuracies:
    plt.plot(fold[1])
plt.title('Validation Accuracy')
plt.legend(['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1, 3, 3)
for fold in fold_accuracies:
    plt.plot(fold[2])
plt.title('Test Accuracy')
plt.legend(['Fold 1', 'Fold 2', 'Fold 3', 'Fold 4', 'Fold 5'])
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()

print("Average Training Accuracy: ", np.mean([fold[0][-1] for fold in fold_accuraci
print("Standard Deviation Training Accuracy: ", np.std([fold[0][-1] for fold in fol

print("Average Validation Accuracy: ", np.mean([fold[1][-1] for fold in fold_accura
print("Standard Deviation Validation Accuracy: ", np.std([fold[1][-1] for fold in f

print("Average Test Accuracy: ", np.mean([fold[2][-1] for fold in fold_accuracies])
print("Standard Deviation Test Accuracy: ", np.std([fold[2][-1] for fold in fold_ac
```




```
Average Training Accuracy:  0.8228392919125304
Standard Deviation Training Accuracy:  0.004069240094343042
Average Validation Accuracy:  0.8230555555555557
Standard Deviation Validation Accuracy:  0.0058001702827280965
Average Test Accuracy:  0.842386185243328
Standard Deviation Test Accuracy:  0.002737142193746124
```

For K-fold cross validation, where we use 5 folds, we can see that the training and validation loss and accuracy curves are quite similar across the folds. This indicates that the model is generalizing well to the unseen data. The average and standard deviation of the losses and accuracies are also consistent across the folds.

Precision, Recall, F1 Score

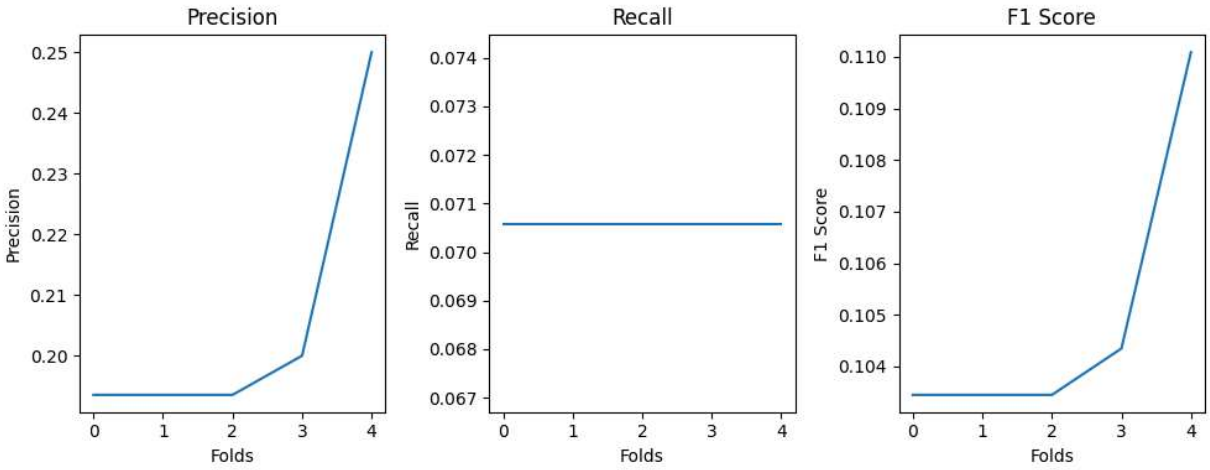
In [382…

```
# average and standard deviation of precision, recall, f1 score

precisions = []
recalls = []
f1s = []

for i in range(folds):
    TP = np.sum((Y_test == 1) & (sigmoid(np.dot(X_test_scaled, fold_Ws[i]) + fold_b
    TN = np.sum((Y_test == 0) & (sigmoid(np.dot(X_test_scaled, fold_Ws[i]) + fold_b
    FP = np.sum((Y_test == 0) & (sigmoid(np.dot(X_test_scaled, fold_Ws[i]) + fold_b
    FN = np.sum((Y_test == 1) & (sigmoid(np.dot(X_test_scaled, fold_Ws[i]) + fold_b

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0

    precisions.append(precision)
    recalls.append(recall)
    f1s.append(f1)
```

In [383…

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 3, 1)
plt.plot(precisions)
plt.title('Precision')
plt.xlabel('Folds')
plt.ylabel('Precision')

plt.subplot(1, 3, 2)
plt.plot(recalls)
plt.title('Recall')
plt.xlabel('Folds')
plt.ylabel('Recall')

plt.subplot(1, 3, 3)
plt.plot(f1s)
plt.title('F1 Score')
plt.xlabel('Folds')
plt.ylabel('F1 Score')

plt.tight_layout()
plt.show()

print("Average Precision: ", np.mean(precisions))
print("Standard Deviation Precision: ", np.std(precisions))

print("Average Recall: ", np.mean(recalls))
```

```
print("Standard Deviation Recall: ", np.std(recalls))

print("Average F1 Score: ", np.mean(f1s))
print("Standard Deviation F1 Score: ", np.std(f1s))
```




```
Average Precision:  0.2061290322580645
Standard Deviation Precision:  0.022077340170669257
Average Recall:  0.07058823529411765
Standard Deviation Recall:  0.0
Average F1 Score:  0.1049568793584859
Standard Deviation F1 Score:  0.0025909621864295216
```

Precison and F1-score all show similar deviations across the folds as one would expect since we change the training and validation data across the folds. We notice a consistent recall probably because our dataset is not well balanced and has a large number of class 0 values.

(f) Regularization

Best gradient descent method: Scaled Gradient Descent since it has a more stable convergence with less noise
Best stopping criteria to avoid overfitting: Early Stopping
Implement early stopping using L2 regularization

We must also note that there is not a lot of overfitting so there might not b significant changes in the accuracy and loss but the convergence might be earlier.

Ridge Regularization: L2 Regularization

In [384…

```
def gradient_descent_ridge(X_train, Y_train, X_val, Y_val, X_test, Y_test, learning
    W, b = initialize_weights(X_train.shape[1])

    train_loss = []
    val_loss = []
    test_loss = []
    train_accuracy = []
    val_accuracy = []
```

```
    test_accuracy = []

    Weights = []
    Biases = []

    m = X_train.shape[0]

    for epoch in range(epochs):
        z = np.dot(X_train, W) + b
        y_hat = sigmoid(z)
        loss = cross_entropy_loss(Y_train, y_hat) + alpha * np.sum(W ** 2)

        dW = 1/m * np.dot(X_train.T, (y_hat - Y_train)) + 2 * alpha * W
        db = 1/m * np.sum(y_hat - Y_train)

        W -= learning_rate * dW
        b -= learning_rate * db

        train_loss.append(loss)
        val_loss.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_loss.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test, W) + b))

        Weights.append(W)
        Biases.append(b)

        train_accuracy.append(accuracy(Y_train, y_hat))
        val_accuracy.append(accuracy(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_accuracy.append(accuracy(Y_test, sigmoid(np.dot(X_test, W) + b)))

    return W, b, train_loss, val_loss, test_loss, train_accuracy, val_accuracy, tes
```

In [385…

```
def gradient_descent_scaled_ridge(X_train, Y_train, X_val, Y_val, X_test, Y_test, l
    W, b = initialize_weights(X_train.shape[1])

    train_loss = []
    val_loss = []
    test_loss = []
    train_accuracy = []
    val_accuracy = []
    test_accuracy = []

    Weights = []
    Biases = []

    m = X_train.shape[0]

    for epoch in range(epochs):
        z = np.dot(X_train, W) + b
        y_hat = sigmoid(z)
        loss = cross_entropy_loss(Y_train, y_hat) + alpha * np.sum(W ** 2)

        dW = 1/m * np.dot(X_train.T, (y_hat - Y_train)) + 2 * alpha * W
        db = 1/m * np.sum(y_hat - Y_train)

        W -= learning_rate * dW
        b -= learning_rate * db
```

```
        train_loss.append(loss)
        val_loss.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_loss.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test, W) + b))

        Weights.append(W)
        Biases.append(b)

        train_accuracy.append(accuracy(Y_train, y_hat))
        val_accuracy.append(accuracy(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_accuracy.append(accuracy(Y_test, sigmoid(np.dot(X_test, W) + b)))

    return W, b, train_loss, val_loss, test_loss, train_accuracy, val_accuracy, tes
```

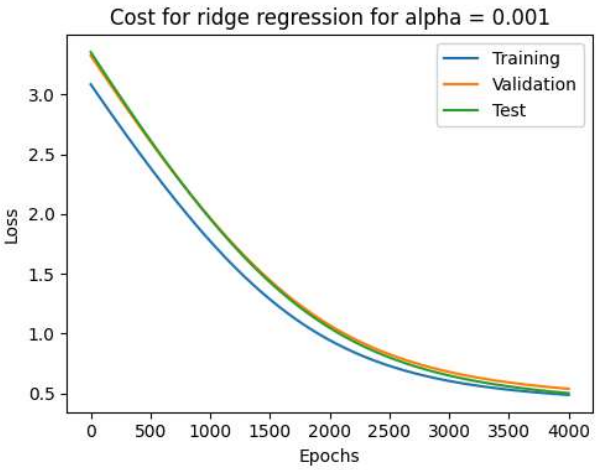
In [387…

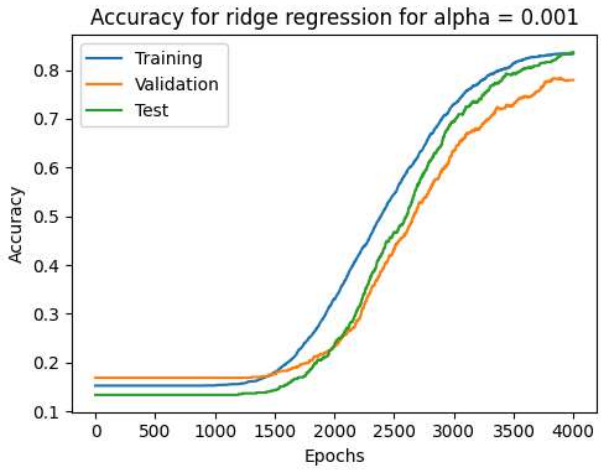
```
alphas = [0.001, 0.01, 0.1, 1, 10]
```

In [391…

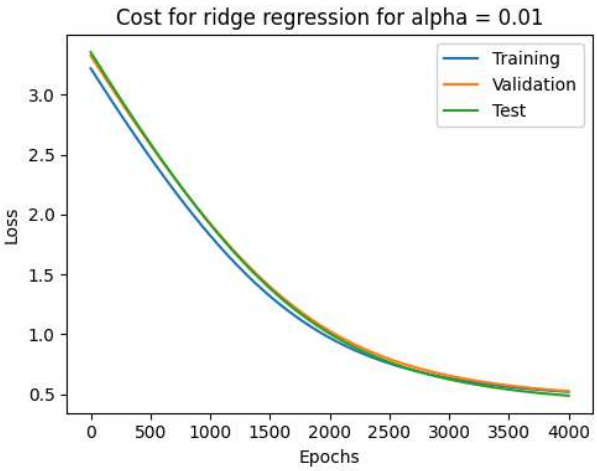
```
for alpha in alphas:
    results_ridge = gradient_descent_ridge(X_train_scaled, Y_train, X_val_scaled, Y
    W = results_ridge[0]
    b = results_ridge[1]
    train_loss = results_ridge[2]
    val_loss = results_ridge[3]
    test_loss = results_ridge[4]
    train_accuracy = results_ridge[5]
    val_accuracy = results_ridge[6]
    test_accuracy = results_ridge[7]
    Weights = results_ridge[8]
    Biases = results_ridge[9]

    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss, label='train loss')
    plt.plot(val_loss, label='val loss')
    plt.plot(test_loss, label='test loss')
    plt.title('Cost for ridge regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracy, label='train accuracy')
    plt.plot(val_accuracy, label='val accuracy')
    plt.plot(test_accuracy, label='test accuracy')
    plt.title('Accuracy for ridge regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')

    plt.tight_layout()
    plt.show()

    print("Training Loss: ", train_loss[-1])
    print("Training Accuracy: ", train_accuracy[-1])
    print("Validation Loss: ", val_loss[-1])
    print("Validation Accuracy: ", val_accuracy[-1])
```

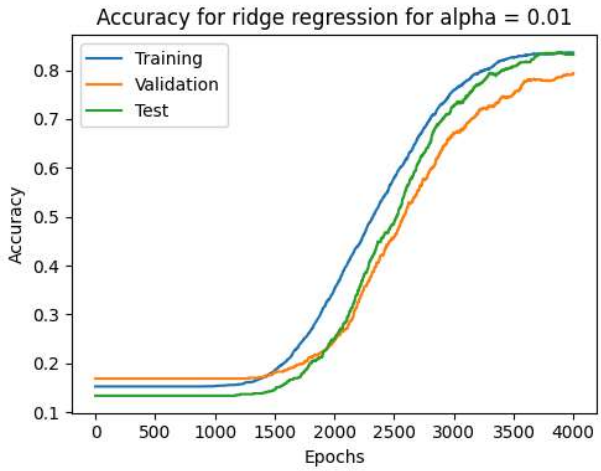
```
    print("Test Loss: ", test_loss[-1])
    print("Test Accuracy: ", test_accuracy[-1])

    print("Weights: ", W)
    print("Bias: ", b)
```



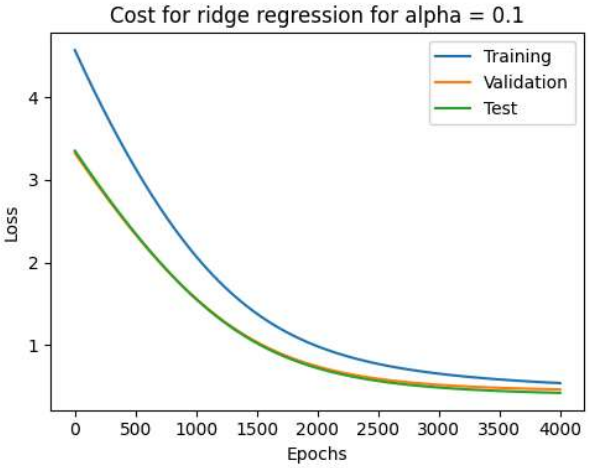


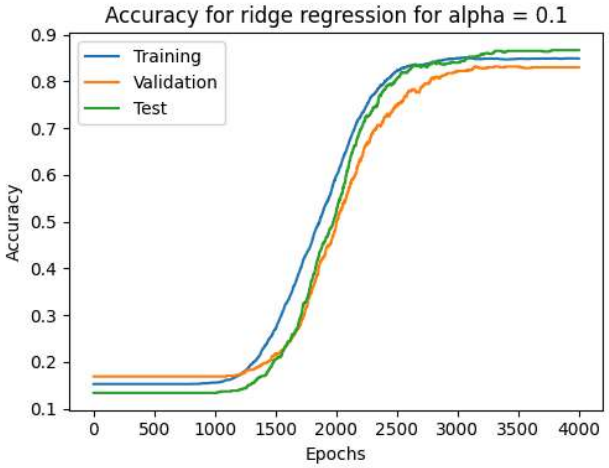

```
Training Loss:  0.486982555860363
Training Accuracy:  0.8334457181389077
Validation Loss:  0.5377457530218459
Validation Accuracy:  0.7795275590551181
Test Loss:  0.5011553467774306
Test Accuracy:  0.8367346938775511
Weights:  [ 0.14594845  0.13305432  0.32092673 -0.01214437  0.73131516  0.93634445
  0.98539259  0.38026961  0.94659409  0.61090033  0.5621255   0.28340699
  0.50681724  0.3443588   0.76651652]
Bias:  -1.941606671386727
```



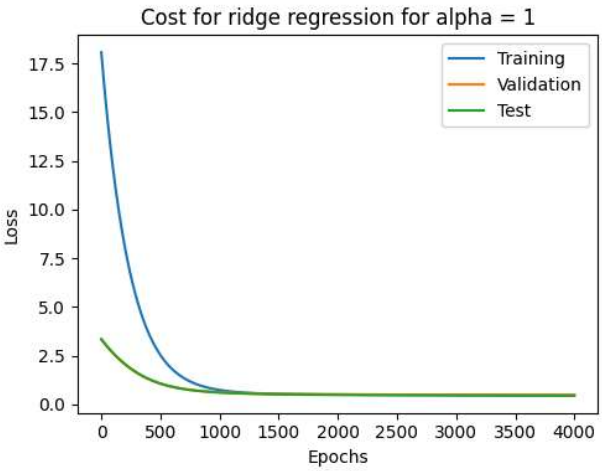


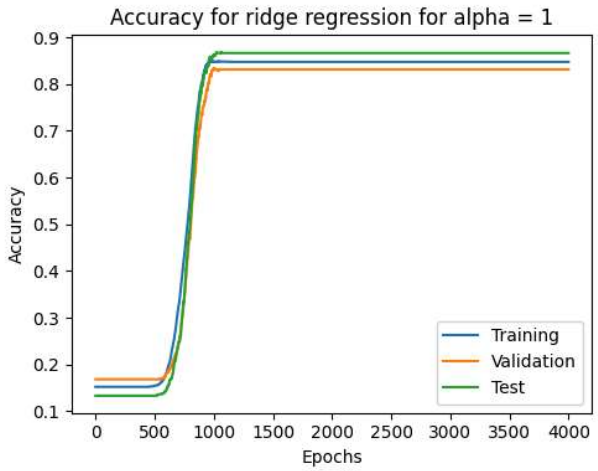

```
Training Loss:  0.518357066773077
Training Accuracy:  0.8361429534726905
Validation Loss:  0.523383079560454
Validation Accuracy:  0.7937007874015748
Test Loss:  0.4863835497506067
Test Accuracy:  0.8320251177394035
Weights:  [ 0.13308392  0.12080601  0.29405239 -0.01624625  0.67947683  0.87117044
  0.9170224   0.35281031  0.88085478  0.5665003   0.52136644  0.26035025
  0.46907347  0.31680318  0.71220076]
Bias:  -1.9017314948813215
```



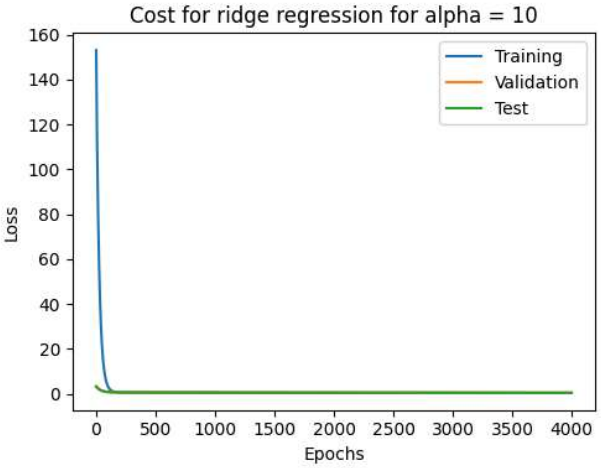


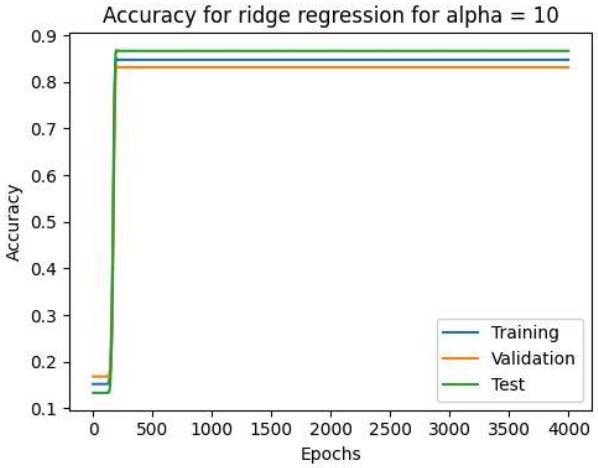

```
Training Loss:  0.5420587969650658
Training Accuracy:  0.8486176668914363
Validation Loss:  0.4659304293357003
Validation Accuracy:  0.8299212598425196
Test Loss:  0.42414055086335517
Test Accuracy:  0.8665620094191523
Weights:  [ 0.04926862  0.04032408  0.11416858 -0.03876821  0.3247058   0.42560086
  0.44732508  0.17066142  0.43089455  0.26330712  0.24386327  0.106196
  0.212425    0.13112016  0.34045853]
Bias:  -1.5947242251033515
```






```
Training Loss:  0.4637415095239325
Training Accuracy:  0.8476062036412677
Validation Loss:  0.47747356061835233
Validation Accuracy:  0.831496062992126
Test Loss:  0.44088193272557763
Test Accuracy:  0.8665620094191523
Weights:  [-0.01580019 -0.01839827 -0.02189559 -0.02625043 -0.00529085  0.00173116
  0.00099654 -0.00375674  0.0019709  -0.01026814 -0.00899544 -0.01759871
 -0.01309717 -0.01873989 -0.00505102]
Bias:  -0.9936499545995667
```



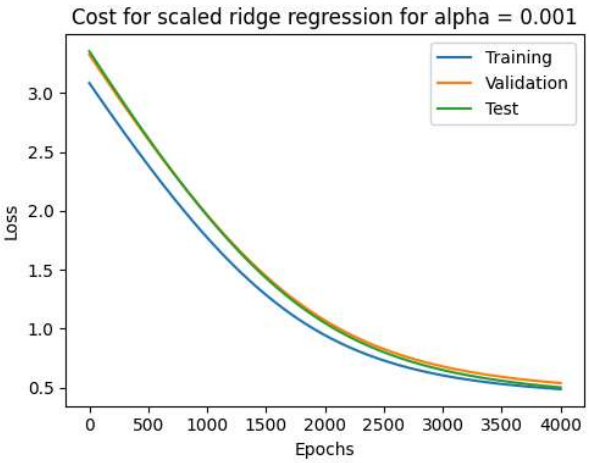


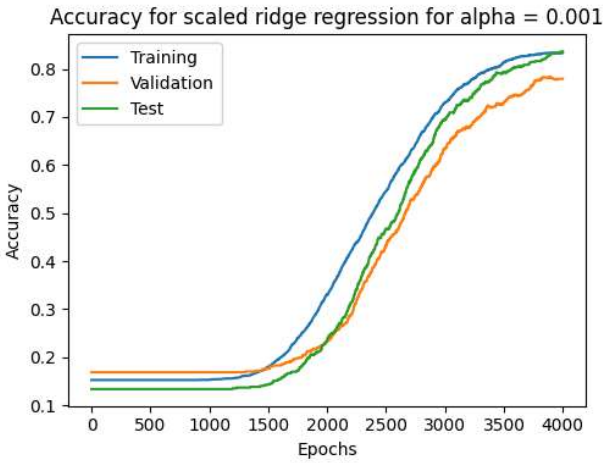

```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```

In [392…

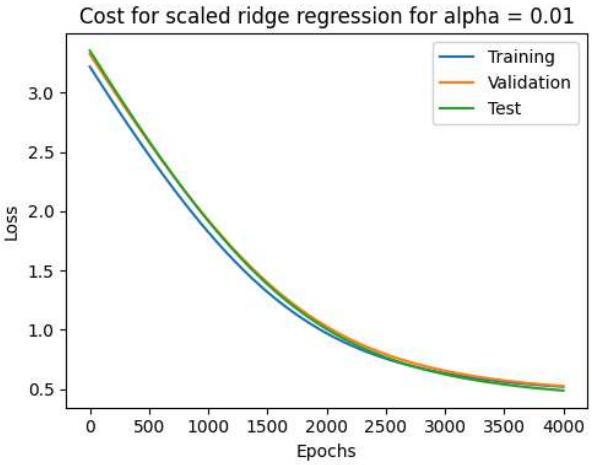
```
for alpha in alphas:
    results_scaled_ridge = gradient_descent_scaled_ridge(X_train_scaled, Y_train, X
    W_scaled = results_scaled_ridge[0]
    b_scaled = results_scaled_ridge[1]
    train_loss_scaled = results_scaled_ridge[2]
    val_loss_scaled = results_scaled_ridge[3]
    test_loss_scaled = results_scaled_ridge[4]
    train_accuracy_scaled = results_scaled_ridge[5]
    val_accuracy_scaled = results_scaled_ridge[6]
    test_accuracy_scaled = results_scaled_ridge[7]
    Weights_scaled = results_scaled_ridge[8]
    Biases_scaled = results_scaled_ridge[9]

    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss_scaled, label='train loss')
    plt.plot(val_loss_scaled, label='val loss')
    plt.plot(test_loss_scaled, label='test loss')
    plt.title('Cost for scaled ridge regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracy_scaled, label='train accuracy')
    plt.plot(val_accuracy_scaled, label='val accuracy')
    plt.plot(test_accuracy_scaled, label='test accuracy')
    plt.title('Accuracy for scaled ridge regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
```

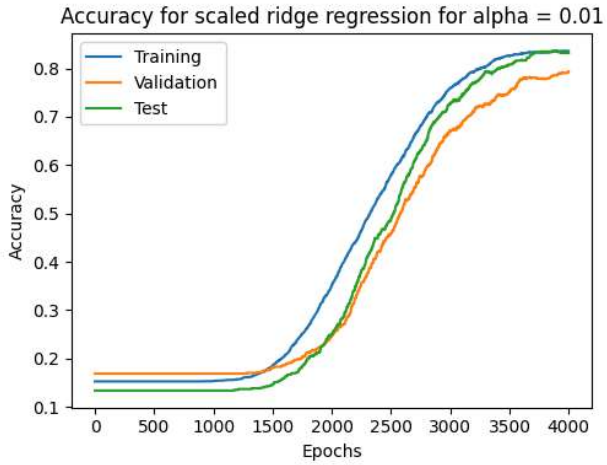
```
    plt.ylabel('Accuracy')

    plt.tight_layout()
    plt.show()

    print("Training Loss: ", train_loss_scaled[-1])
    print("Training Accuracy: ", train_accuracy_scaled[-1])
    print("Validation Loss: ", val_loss_scaled[-1])
    print("Validation Accuracy: ", val_accuracy_scaled[-1])
    print("Test Loss: ", test_loss_scaled[-1])
    print("Test Accuracy: ", test_accuracy_scaled[-1])

    print("Weights: ", W_scaled)
    print("Bias: ", b_scaled)
```



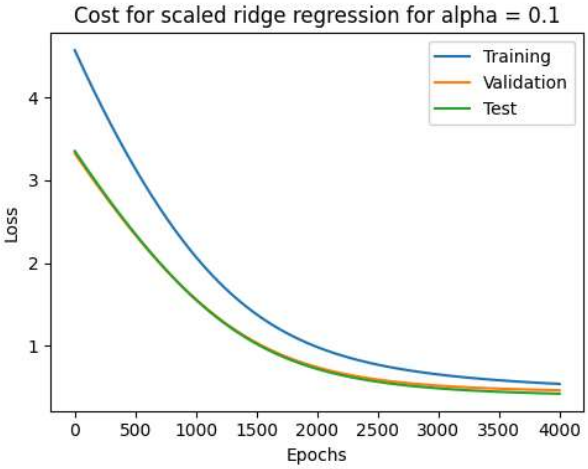


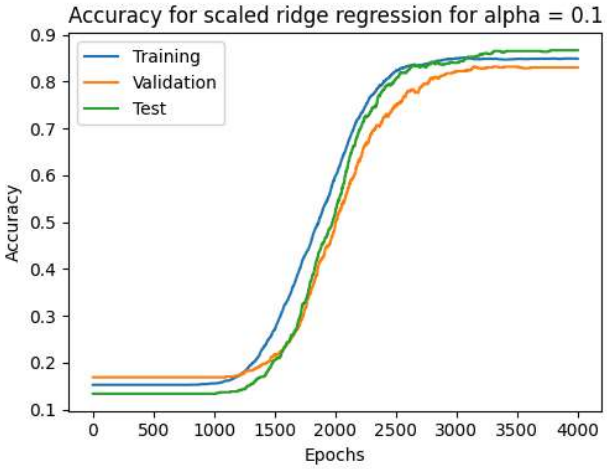

```
Training Loss:  0.486982555860363
Training Accuracy:  0.8334457181389077
Validation Loss:  0.5377457530218459
Validation Accuracy:  0.7795275590551181
Test Loss:  0.5011553467774306
Test Accuracy:  0.8367346938775511
Weights:  [ 0.14594845  0.13305432  0.32092673 -0.01214437  0.73131516  0.93634445
  0.98539259  0.38026961  0.94659409  0.61090033  0.5621255   0.28340699
  0.50681724  0.3443588   0.76651652]
Bias:  -1.941606671386727
```



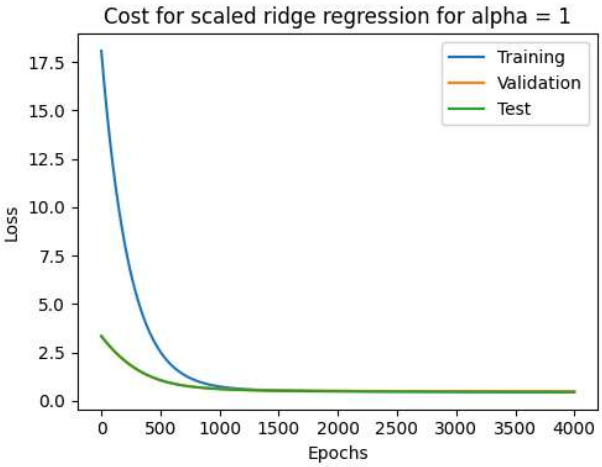


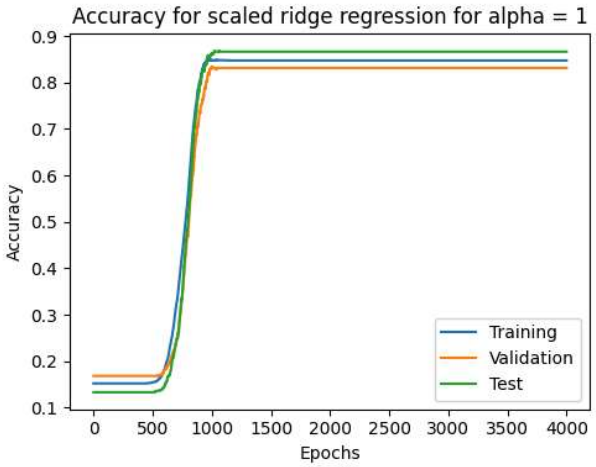
```
Training Loss:  0.518357066773077
Training Accuracy:  0.8361429534726905
Validation Loss:  0.523383079560454
Validation Accuracy:  0.7937007874015748
Test Loss:  0.4863835497506067
Test Accuracy:  0.8320251177394035
Weights:  [ 0.13308392  0.12080601  0.29405239 -0.01624625  0.67947683  0.87117044
  0.9170224   0.35281031  0.88085478  0.5665003   0.52136644  0.26035025
  0.46907347  0.31680318  0.71220076]
Bias:  -1.9017314948813215
```



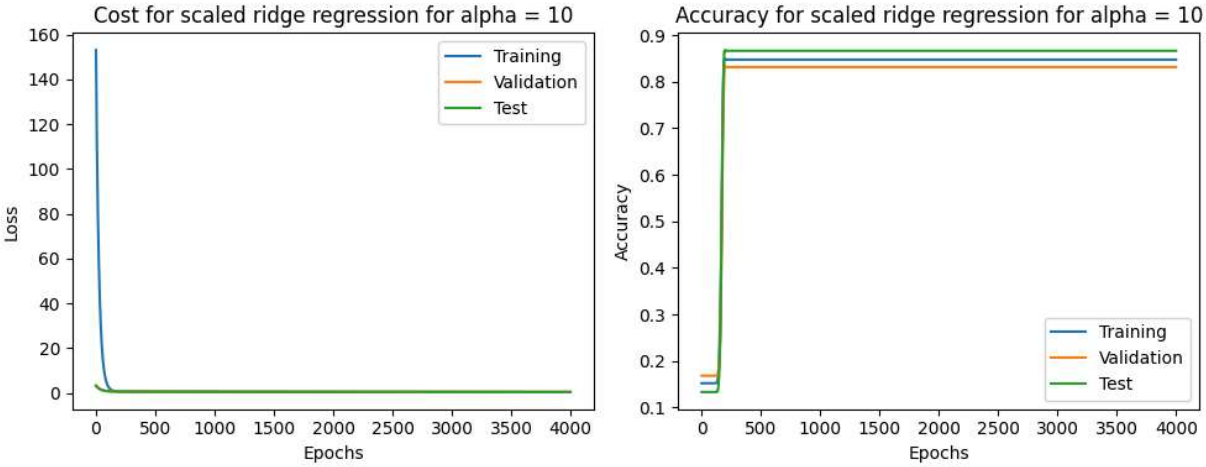



```
Training Loss:  0.5420587969650658
Training Accuracy:  0.8486176668914363
Validation Loss:  0.4659304293357003
Validation Accuracy:  0.8299212598425196
Test Loss:  0.42414055086335517
Test Accuracy:  0.8665620094191523
Weights:  [ 0.04926862  0.04032408  0.11416858 -0.03876821  0.3247058   0.42560086
  0.44732508  0.17066142  0.43089455  0.26330712  0.24386327  0.106196
  0.212425    0.13112016  0.34045853]
Bias:  -1.5947242251033515
```





```
Training Loss:  0.4637415095239325
Training Accuracy:  0.8476062036412677
Validation Loss:  0.47747356061835233
Validation Accuracy:  0.831496062992126
Test Loss:  0.44088193272557763
Test Accuracy:  0.8665620094191523
Weights:  [-0.01580019 -0.01839827 -0.02189559 -0.02625043 -0.00529085  0.00173116
  0.00099654 -0.00375674  0.0019709  -0.01026814 -0.00899544 -0.01759871
 -0.01309717 -0.01873989 -0.00505102]
Bias:  -0.9936499545995667
```




```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```

As we increase the value of alpha, we notice a better and improved convergence as well as strong effects of early stopping compared to our original graphs. This is because the regularization term penalizes the weights more heavily and hence the model converges faster. However, such large values of alpha might lead to underfitting.

Lasso Regularization: L1 Regularization

In [393…

```
def gradient_descent_lasso(X_train, Y_train, X_val, Y_val, X_test, Y_test, learning
    W, b = initialize_weights(X_train.shape[1])

    train_loss = []
    val_loss = []
    test_loss = []
    train_accuracy = []
    val_accuracy = []
    test_accuracy = []
```

```
    Weights = []
    Biases = []

    m = X_train.shape[0]

    for epoch in range(epochs):
        z = np.dot(X_train, W) + b
        y_hat = sigmoid(z)
        loss = cross_entropy_loss(Y_train, y_hat) + alpha * np.sum(np.abs(W))

        dW = 1/m * np.dot(X_train.T, (y_hat - Y_train)) + alpha * np.sign(W)
        db = 1/m * np.sum(y_hat - Y_train)

        W -= learning_rate * dW
        b -= learning_rate * db

        train_loss.append(loss)
        val_loss.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_loss.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test, W) + b))

        Weights.append(W)
        Biases.append(b)

        train_accuracy.append(accuracy(Y_train, y_hat))
        val_accuracy.append(accuracy(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_accuracy.append(accuracy(Y_test, sigmoid(np.dot(X_test, W) + b)))

    return W, b, train_loss, val_loss, test_loss, train_accuracy, val_accuracy, tes
```

In [394…

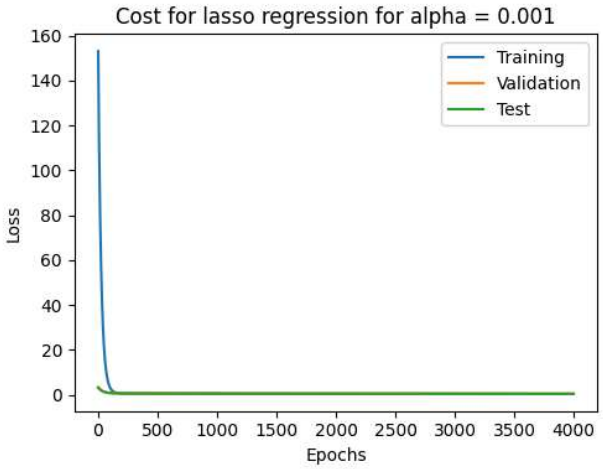
```
def gradient_descent_scaled_lasso(X_train, Y_train, X_val, Y_val, X_test, Y_test, l
    W, b = initialize_weights(X_train.shape[1])

    train_loss = []
    val_loss = []
    test_loss = []
    train_accuracy = []
    val_accuracy = []
    test_accuracy = []

    Weights = []
    Biases = []

    m = X_train.shape[0]

    for epoch in range(epochs):
        z = np.dot(X_train, W) + b
        y_hat = sigmoid(z)
        loss = cross_entropy_loss(Y_train, y_hat) + alpha * np.sum(np.abs(W))

        dW = 1/m * np.dot(X_train.T, (y_hat - Y_train)) + alpha * np.sign(W)
        db = 1/m * np.sum(y_hat - Y_train)

        W -= learning_rate * dW
        b -= learning_rate * db

        train_loss.append(loss)
```

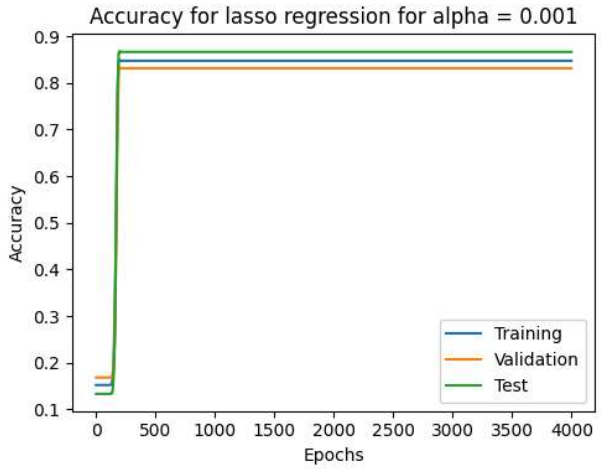
```
        val_loss.append(cross_entropy_loss(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_loss.append(cross_entropy_loss(Y_test, sigmoid(np.dot(X_test, W) + b))

        Weights.append(W)
        Biases.append(b)

        train_accuracy.append(accuracy(Y_train, y_hat))
        val_accuracy.append(accuracy(Y_val, sigmoid(np.dot(X_val, W) + b)))
        test_accuracy.append(accuracy(Y_test, sigmoid(np.dot(X_test, W) + b)))

    return W, b, train_loss, val_loss, test_loss, train_accuracy, val_accuracy, tes
```

In [395…

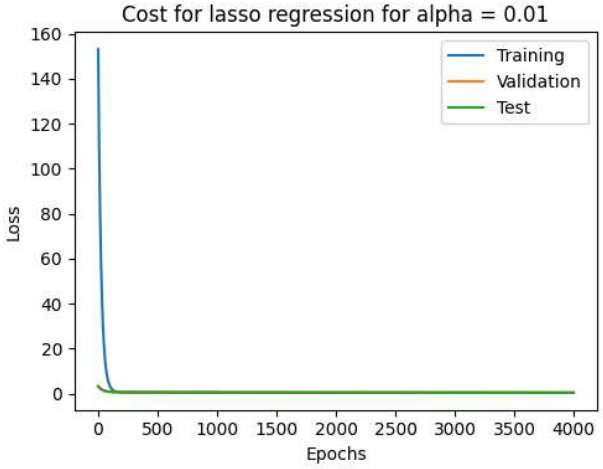
```
for alpha in alphas:
    results_lasso = gradient_descent_lasso(X_train_scaled, Y_train, X_val_scaled, Y
    W_lasso = results_lasso[0]
    b_lasso = results_lasso[1]
    train_loss_lasso = results_lasso[2]
    val_loss_lasso = results_lasso[3]
    test_loss_lasso = results_lasso[4]
    train_accuracy_lasso = results_lasso[5]
    val_accuracy_lasso = results_lasso[6]
    test_accuracy_lasso = results_lasso[7]
    Weights_lasso = results_lasso[8]
    Biases_lasso = results_lasso[9]

    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss, label='train loss')
    plt.plot(val_loss, label='val loss')
    plt.plot(test_loss, label='test loss')
    plt.title('Cost for lasso regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracy, label='train accuracy')
    plt.plot(val_accuracy, label='val accuracy')
    plt.plot(test_accuracy, label='test accuracy')
    plt.title('Accuracy for lasso regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')

    plt.tight_layout()
    plt.show()

    print("Training Loss: ", train_loss[-1])
    print("Training Accuracy: ", train_accuracy[-1])
    print("Validation Loss: ", val_loss[-1])
    print("Validation Accuracy: ", val_accuracy[-1])
    print("Test Loss: ", test_loss[-1])
    print("Test Accuracy: ", test_accuracy[-1])
```

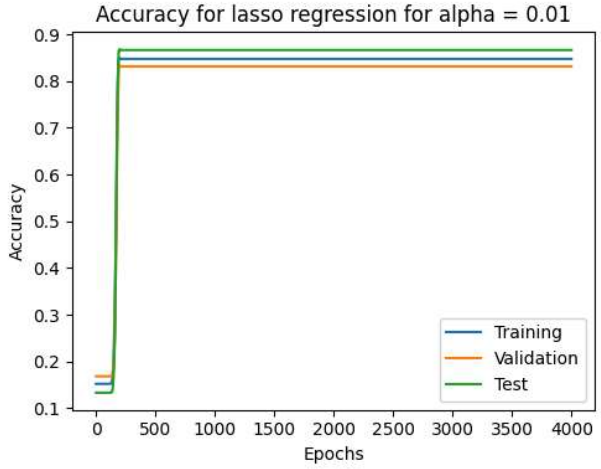
```
    print("Weights: ", W)
    print("Bias: ", b)
```



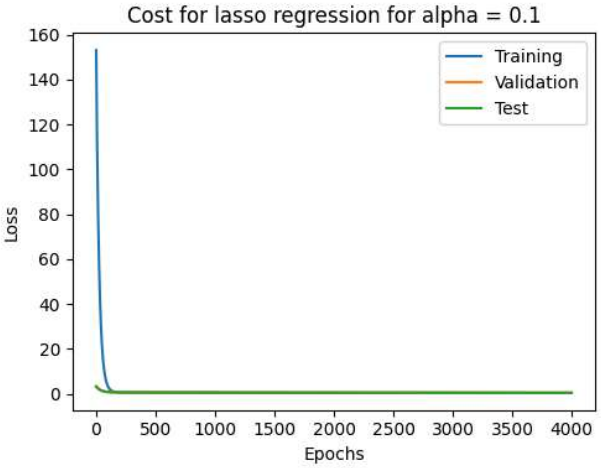


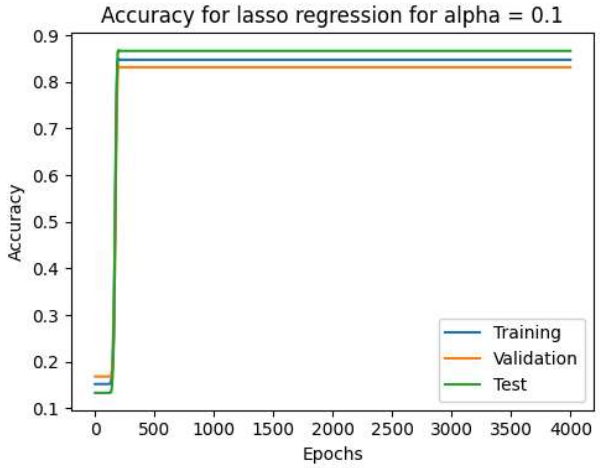

```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```



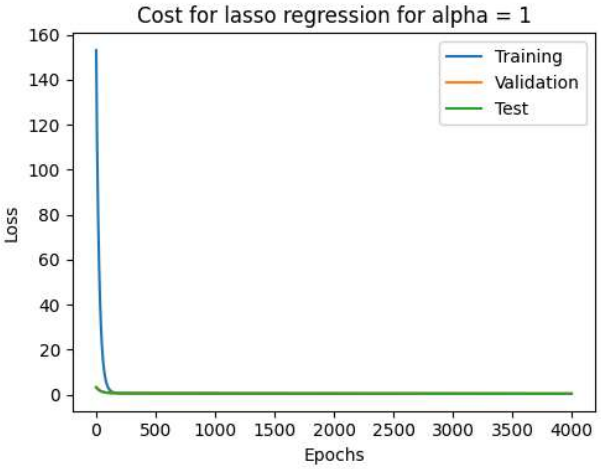


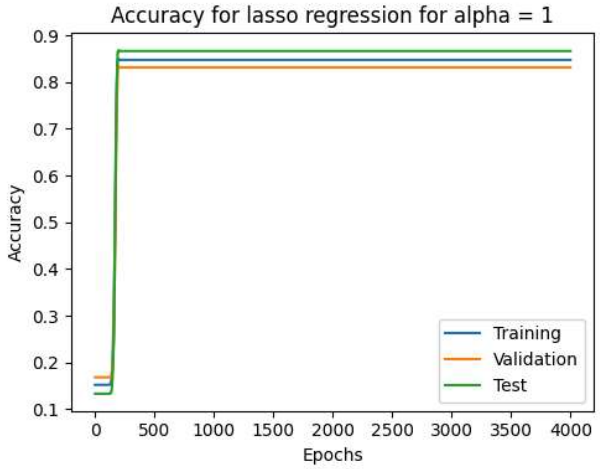

```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```



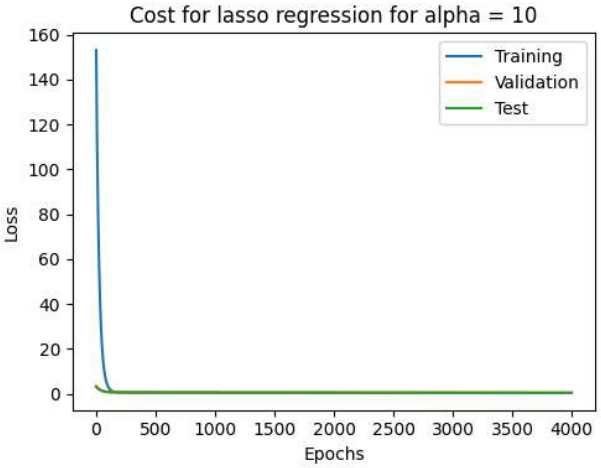


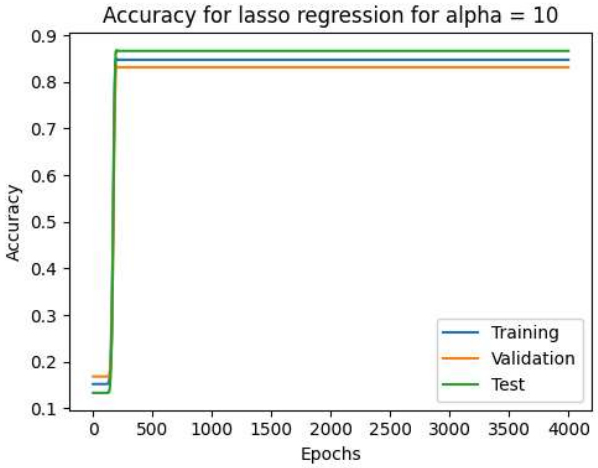

```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```






```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```



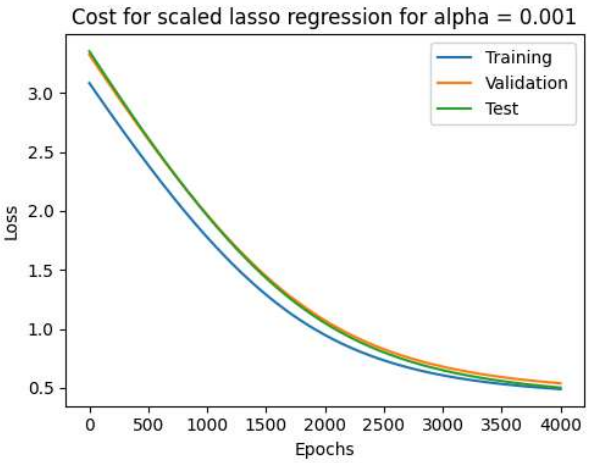


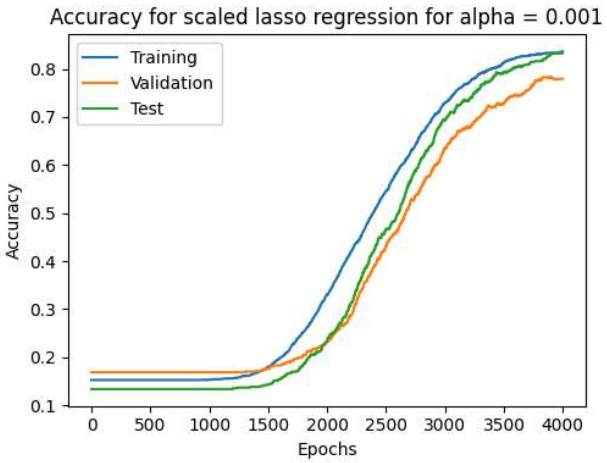

```
Training Loss:  0.4778055554036806
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49193412687254945
Validation Accuracy:  0.831496062992126
Test Loss:  0.4601359718404014
Test Accuracy:  0.8665620094191523
Weights:  [-2.05822385e-03 -2.30709545e-03 -2.53701905e-03 -3.17945478e-03
 -7.00207598e-04  1.13777496e-04  6.18451259e-05 -7.06591063e-04
  1.40687552e-04 -1.24368568e-03 -1.14412684e-03 -2.13610860e-03
 -1.57483319e-03 -2.21709869e-03 -6.48877751e-04]
Bias:  -0.9007296605597906
```

In [396…

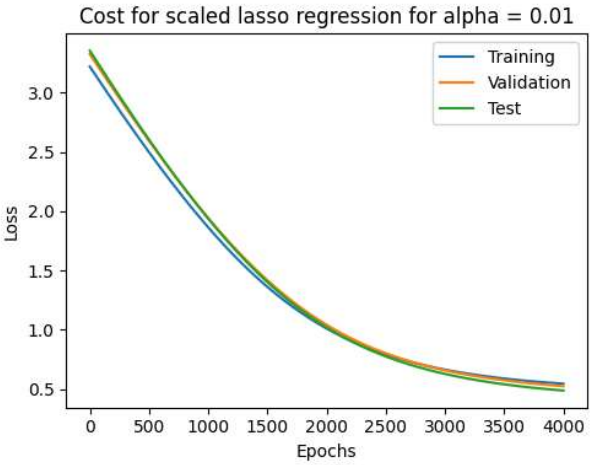
```
for alpha in alphas:
    results_scaled_lasso = gradient_descent_scaled_lasso(X_train_scaled, Y_train, X
    W_scaled_lasso = results_scaled_lasso[0]
    b_scaled_lasso = results_scaled_lasso[1]
    train_loss_scaled_lasso = results_scaled_lasso[2]
    val_loss_scaled_lasso = results_scaled_lasso[3]
    test_loss_scaled_lasso = results_scaled_lasso[4]
    train_accuracy_scaled_lasso = results_scaled_lasso[5]
    val_accuracy_scaled_lasso = results_scaled_lasso[6]
    test_accuracy_scaled_lasso = results_scaled_lasso[7]
    Weights_scaled_lasso = results_scaled_lasso[8]
    Biases_scaled_lasso = results_scaled_lasso[9]

    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss_scaled_lasso, label='train loss')
    plt.plot(val_loss_scaled_lasso, label='val loss')
    plt.plot(test_loss_scaled_lasso, label='test loss')
    plt.title('Cost for scaled lasso regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracy_scaled_lasso, label='train accuracy')
    plt.plot(val_accuracy_scaled_lasso, label='val accuracy')
    plt.plot(test_accuracy_scaled_lasso, label='test accuracy')
    plt.title('Accuracy for scaled lasso regression for alpha = ' + str(alpha))
    plt.legend(['Training', 'Validation', 'Test'])
    plt.xlabel('Epochs')
```

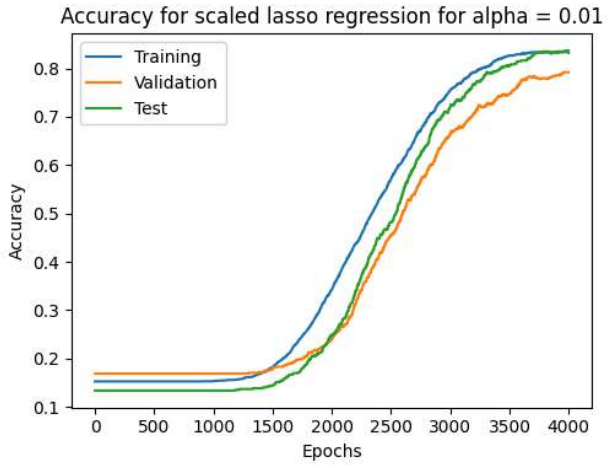
```
    plt.ylabel('Accuracy')

    plt.tight_layout()
    plt.show()

    print("Training Loss: ", train_loss_scaled_lasso[-1])
    print("Training Accuracy: ", train_accuracy_scaled_lasso[-1])
    print("Validation Loss: ", val_loss_scaled_lasso[-1])
    print("Validation Accuracy: ", val_accuracy_scaled_lasso[-1])
    print("Test Loss: ", test_loss_scaled_lasso[-1])
    print("Test Accuracy: ", test_accuracy_scaled_lasso[-1])

    print("Weights: ", W_scaled_lasso)
    print("Bias: ", b_scaled_lasso)
```



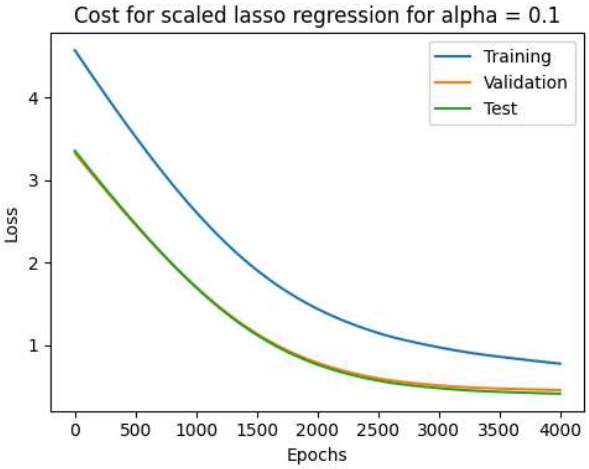


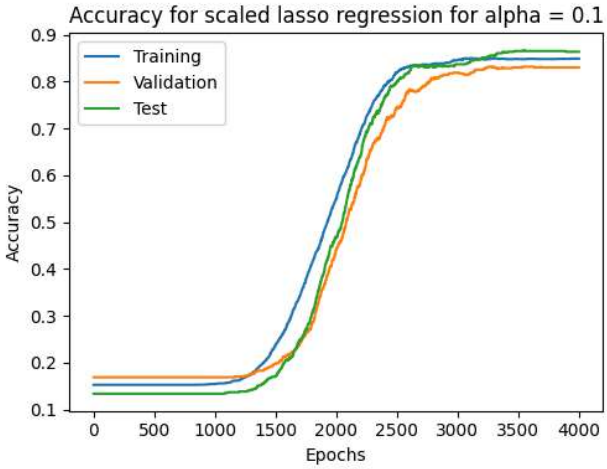

```
Training Loss:  0.48940845248860343
Training Accuracy:  0.8334457181389077
Validation Loss:  0.5379321800448056
Validation Accuracy:  0.7795275590551181
Test Loss:  0.5013377492461367
Test Accuracy:  0.8367346938775511
Weights:  [ 0.14525377  0.13218723  0.32125018 -0.01334728  0.73385779  0.94000539
  0.98931912  0.38083679  0.95030362  0.61275125  0.56369445  0.28344206
  0.50809469  0.34474569  0.76922672]
Bias:  -1.9426387368113978
```



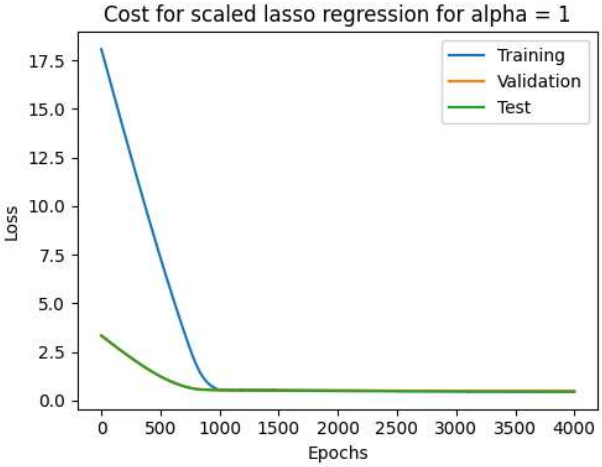


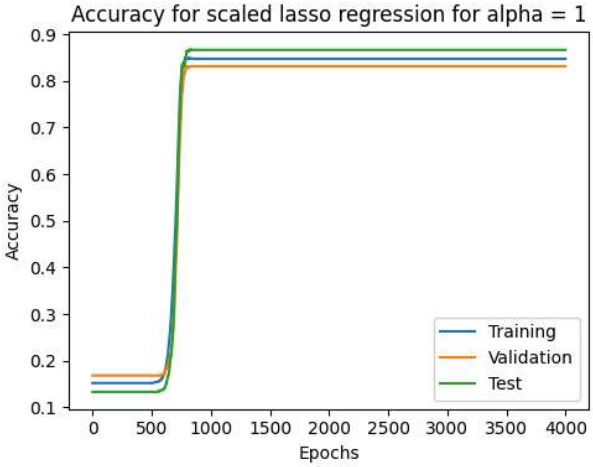
```
Training Loss:  0.5457948384945649
Training Accuracy:  0.8364801078894134
Validation Loss:  0.524664434931338
Validation Accuracy:  0.7921259842519685
Test Loss:  0.48765317486174203
Test Accuracy:  0.8320251177394035
Weights:  [ 0.12546532  0.11151515  0.2960742  -0.02507149  0.7027425   0.90509775
  0.95350784  0.35722714  0.91525217  0.58314717  0.5353287   0.25964616
  0.48023732  0.31944695  0.73705695]
Bias:  -1.9110265410158902
```



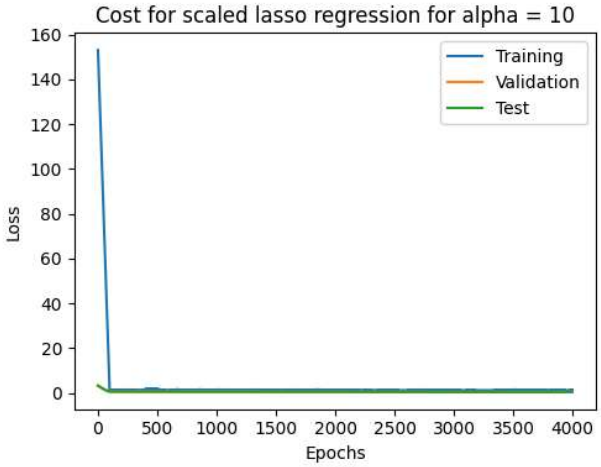


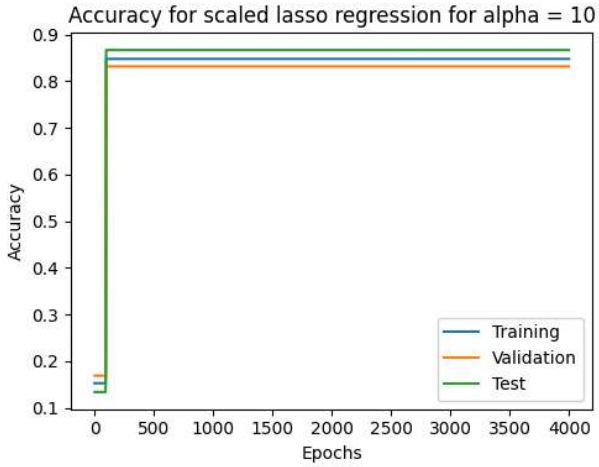

```
Training Loss:  0.7779408474180668
Training Accuracy:  0.8486176668914363
Validation Loss:  0.4580769503726508
Validation Accuracy:  0.8299212598425196
Test Loss:  0.4145951103688879
Test Accuracy:  0.8634222919937206
Weights:  [ 2.73249142e-05 -1.03941955e-05  3.80590564e-02 -1.16639533e-04
  3.87220517e-01  5.57828839e-01  5.95718944e-01  1.19771240e-01
  5.66165676e-01  2.83260118e-01  2.48086618e-01  1.52643650e-02
  1.97103826e-01  5.98230753e-02  4.13576567e-01]
Bias:  -1.6159221997693836
```





```
Training Loss:  0.4736790150095857
Training Accuracy:  0.8476062036412677
Validation Loss:  0.48076754522982224
Validation Accuracy:  0.831496062992126
Test Loss:  0.4453464235614368
Test Accuracy:  0.8665620094191523
Weights:  [-6.78323698e-04 -1.58277346e-04 -3.09905426e-04  4.60400216e-04
  2.55229073e-04 -4.64887449e-04  1.72792808e-04 -7.10075164e-04
 -5.04938671e-04  3.62360320e-05 -8.54340037e-04  1.22673408e-04
  7.84453115e-04 -3.20363969e-04 -7.51008429e-05]
Bias:  -1.0098742875919615
```






```
Training Loss:  1.3298285709642867
Training Accuracy:  0.8476062036412677
Validation Loss:  0.49341614872023026
Validation Accuracy:  0.831496062992126
Test Loss:  0.4620209479410335
Test Accuracy:  0.8665620094191523
Weights:  [-0.00067921  0.00732113  0.00188976  0.00582616 -0.00285673 -0.00135517
  0.00310072 -0.00793749  0.00193161  0.00938149  0.0071607  -0.00271354
  0.00402341  0.00330253  0.00505443]
Bias:  -0.9057295197050361
```

For L1 regularization we see a more faster convergence for unscaled gradient descent in comparison to L2 regularisation. For scaled gradient descent the convergence for the different values of alpha is similar for both regularization methods. This is because L1 regularization is more effective in reducing the number of features by removing irrelevant features and hence the weights converge faster. However large values of alpha might result in loss of important features.

In [ ]:

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import umap
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import RFE
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import GradientBoostingRegressor
```

Loading and preprocessing data

In [2]:
```
data_frame = pd.read_csv('Electricity BILL.csv')
data_frame
```

Out[2]:

	Building_Type	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM
0	Residential	1989	12	50.00000
1	Institutional	1980	6	225.75910
2	Industrial	2006	10	98.75592
3	Commercial	1985	1	68.47069
4	Industrial	2006	12	50.00000
...	...	...	...	.
1245	Residential	1985	10	147.61331
1246	Commercial	2007	5	50.00000
1247	Commercial	1990	1	50.00000
1248	Institutional	2021	6	250.00000
1249	Residential	2017	8	143.82115

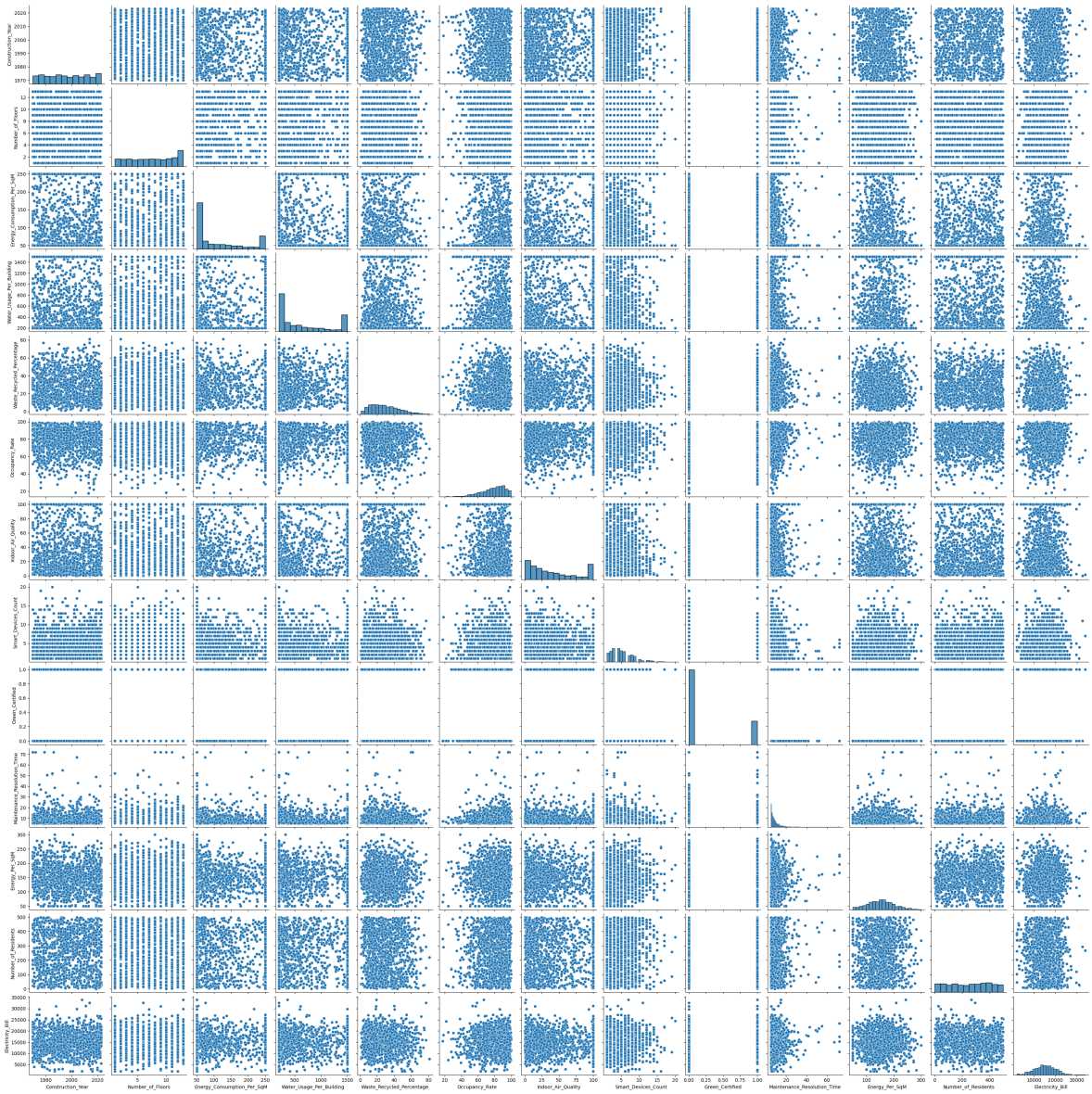
1250 rows × 16 columns

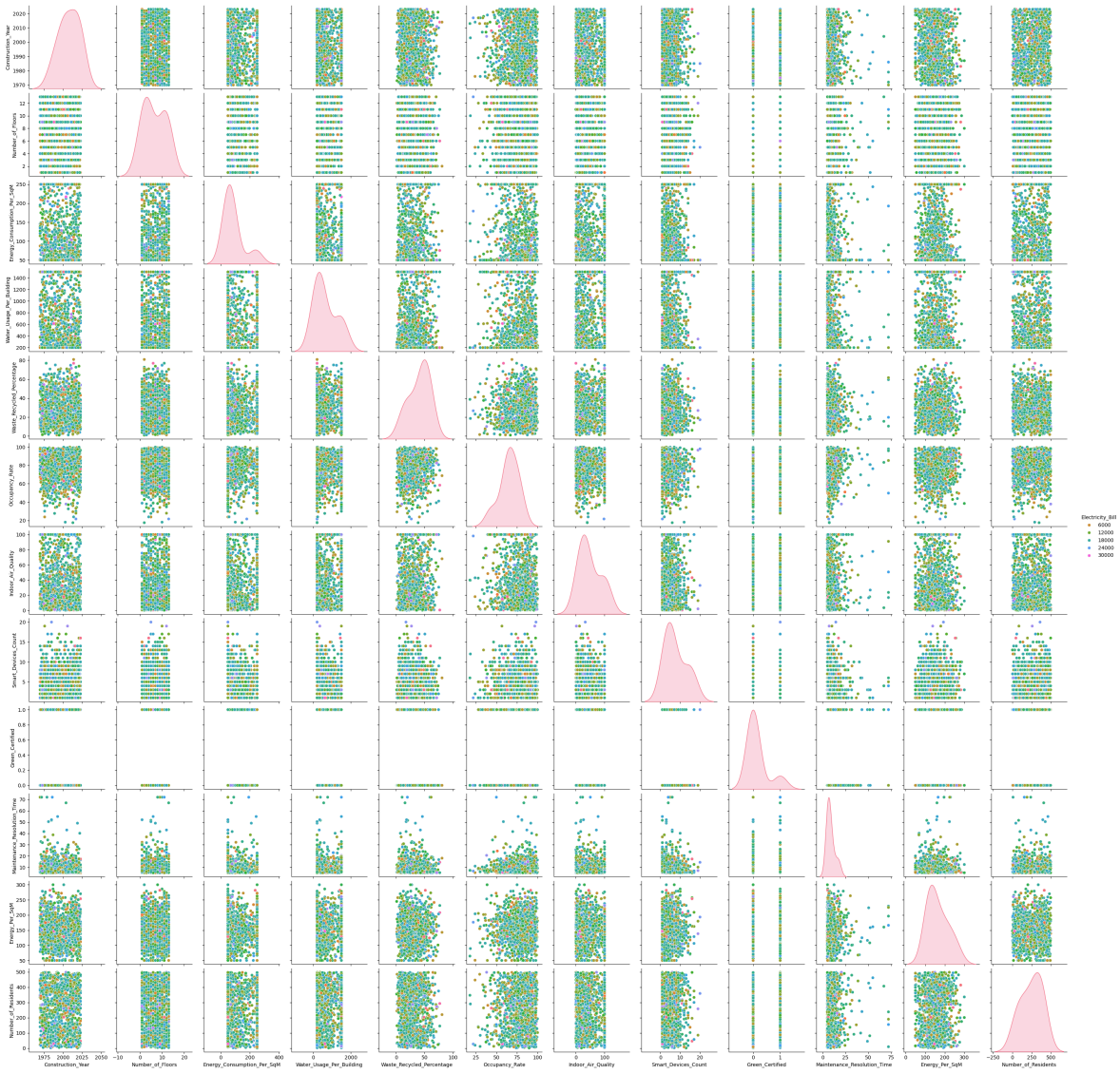
(a) EDA

PairPlot

- Pairplot is a great way to visualize the relationship between all the features in the dataset.
- Diagonal plots are univariate plots and the rest are bivariate plots.

In [3]:
```
# Perform EDA by creating pair plots, box plots, violin plots, count plots for c
# Pair plot
# plt.figsize=(20,10)
sns.pairplot(data_frame)
sns.pairplot(data_frame, hue = 'Electricity_Bill', palette = 'husl')
plt.show()
```





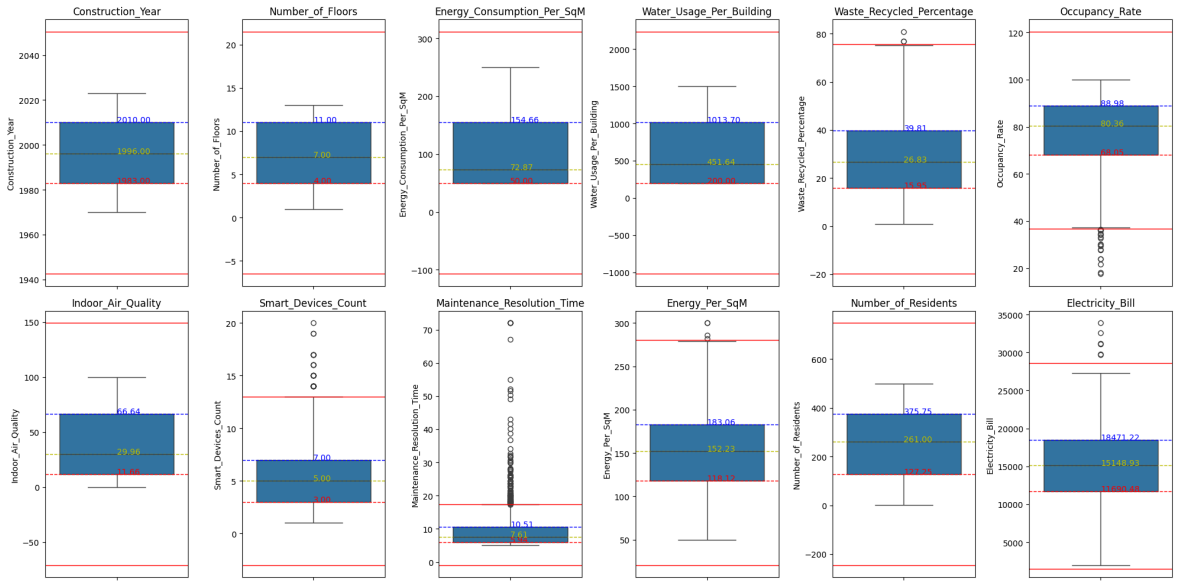
In [4]:
```
numeric_df = data_frame.select_dtypes(include=[np.number])
categorical_df = data_frame.select_dtypes(include=[object])
numeric_df1 = numeric_df.drop('Green_Certified', axis=1)
data_frame_tmp = data_frame.copy()
data_frame_tmp['Green_Certified'] = data_frame_tmp['Green_Certified'].astype(str
categorical_df1 = data_frame_tmp.select_dtypes(include=['object'])
```

Boxplot

- Boxplot is a great way to visualize the distribution of the data.
- It shows the median, quartiles, and outliers.
- It displays the following:
  - Median (Q2/50th Percentile): the middle value of the dataset.
  - First quartile (Q1/25th Percentile): the middle number between the smallest number (not the “minimum”) and the median of the dataset.
  - Third quartile (Q3/75th Percentile): the middle value between the median and the highest value (not the “maximum”) of the dataset.
  - Interquartile range (IQR): 25th to the 75th percentile.
  - Whiskers (shown in blue)
  - Outliers (shown as green points)

In [39]:
```
def calculate_boxplot_stats(data_frame, column):
    q1 = data_frame[column].quantile(0.25)
    q2 = data_frame[column].quantile(0.5)
    q3 = data_frame[column].quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    return q1, q2, q3, iqr, lower_bound, upper_bound
```

In [44]:
```
# Box plot
plt.figure(figsize=(20, 10))
for i, column in enumerate(numeric_df1.columns, 1):
    plt.subplot(2, 6, i)
    sns.boxplot(y=column, data=data_frame, orient='v')

    stats = calculate_boxplot_stats(data_frame, column)

    plt.text(0, stats[0], f'{stats[0]:.2f}', color='r')
    plt.text(0, stats[1], f'{stats[1]:.2f}', color='y')
    plt.text(0, stats[2], f'{stats[2]:.2f}', color='b')


    plt.axhline(stats[0], color='r', linestyle='--', linewidth=1)
    plt.axhline(stats[1], color='y', linestyle='--', linewidth=1)
    plt.axhline(stats[2], color='b', linestyle='--', linewidth=1)
    plt.axhline(stats[4], color='r', linestyle='-', linewidth=1)
    plt.axhline(stats[5], color='r', linestyle='-', linewidth=1)

    plt.title(column)

plt.tight_layout()
plt.show()
```



Inferences from the Boxplots

1. Maintenance_Resolution_Time:

    - Very concentrated spread indicating that most of the data is closely clustered together.

- contains the highest number of outliers indicating presence of extreme cases.

2. Construction_Year and Number_of_Residents:
    - Symmetrical distribution of data with no outliers can be noticed for both cases since the whiskers are of equal length.
    - Presence of no outliers indicate that most buildings were constructed in the same year and have typically the same number of residents.
3. Wasted_Recycled_Percentage, Energy_per_SqM and Electricity_Bill:
    - Almost symmetrical distribution of data with few outliers can be noticed for all three cases.
    - This indicates that although most houses use same amount of energy, electricty and generate same amount of waste, there are few houses that are outliers and use more energy, electricity and generate more waste.
4. Energy Consumption per Sqm, Water Usage Per Building, and Indoor Air Quality:
    - Positive skewness that is the upper whisker is longer than the lower whisker and median is closer to the lower quartile.
    - This indicates that although most values lie in the lower range, there are few houses that use more energy, water and have poor indoor air quality and stretch the whisker upwards.
5. Occupancy_Rate:
    - Large number of outliers below the lower whisker indicating that a large number of houses have a low occupancy rate.

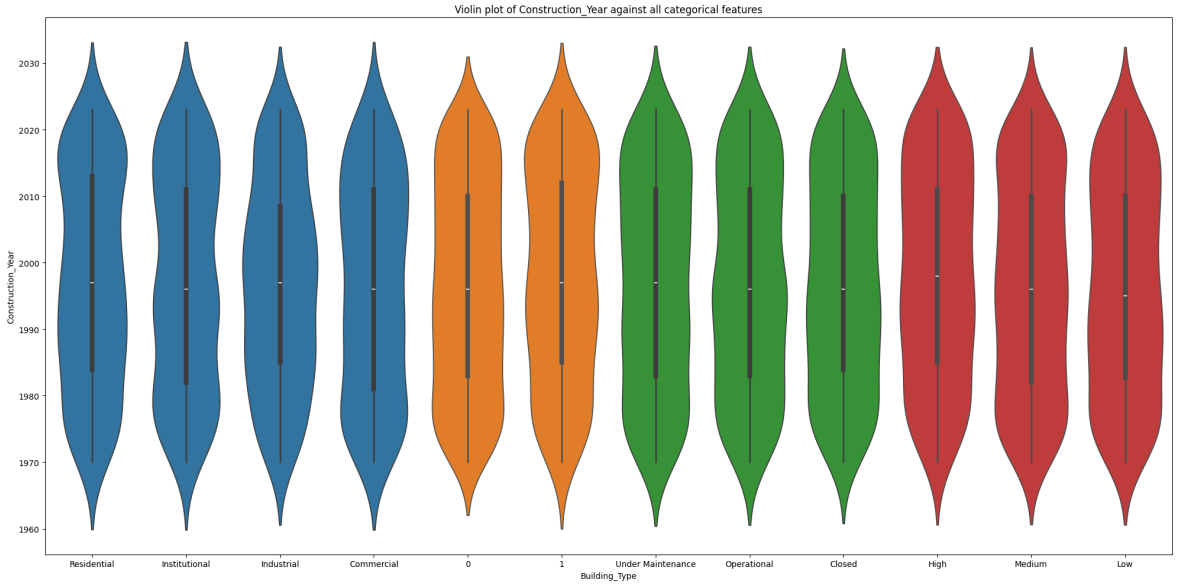
Violinplot

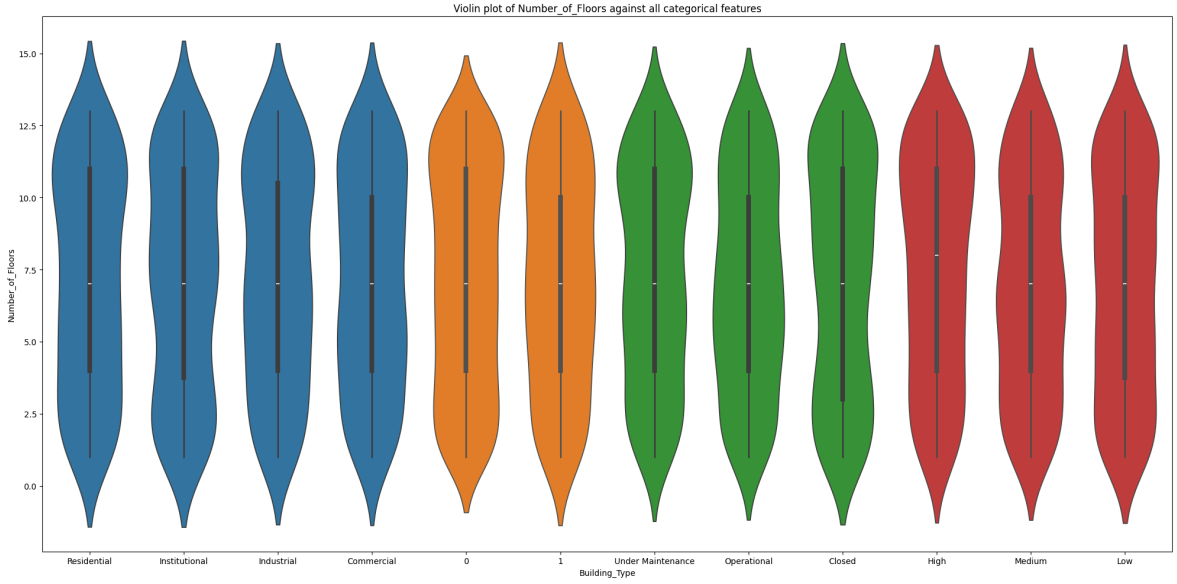
- Violinplot is a combination of a boxplot and a KDE plot.
- It shows the distribution of the data.
- It displays the following:
    - Median (Q2/50th Percentile): the middle value of the dataset.
    - First quartile (Q1/25th Percentile): the middle number between the smallest number (not the “minimum”) and the median of the dataset.
    - Third quartile (Q3/75th Percentile): the middle value between the median and the highest value (not the “maximum”) of the dataset.
    - Interquartile range (IQR): 25th to the 75th percentile.
    - Whiskers (shown in blue)
    - Outliers (shown as green points)
    - KDE plot (shown in orange)
- White centered dot: median
- Thick black line: interquartile range
- Thin black line: 95% confidence interval

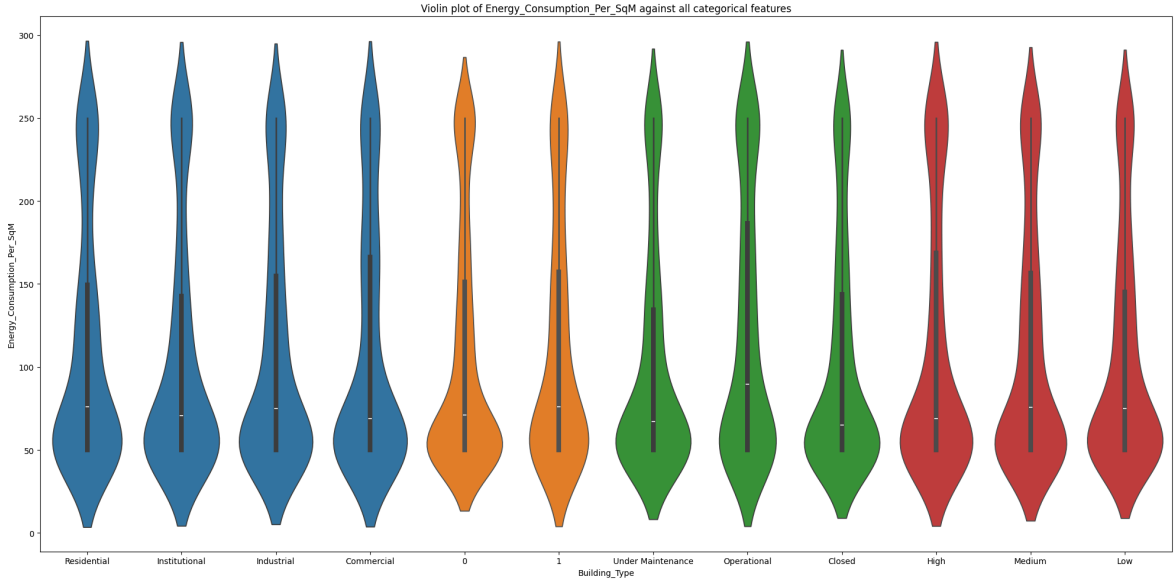
- Line boundary: distribution of the data

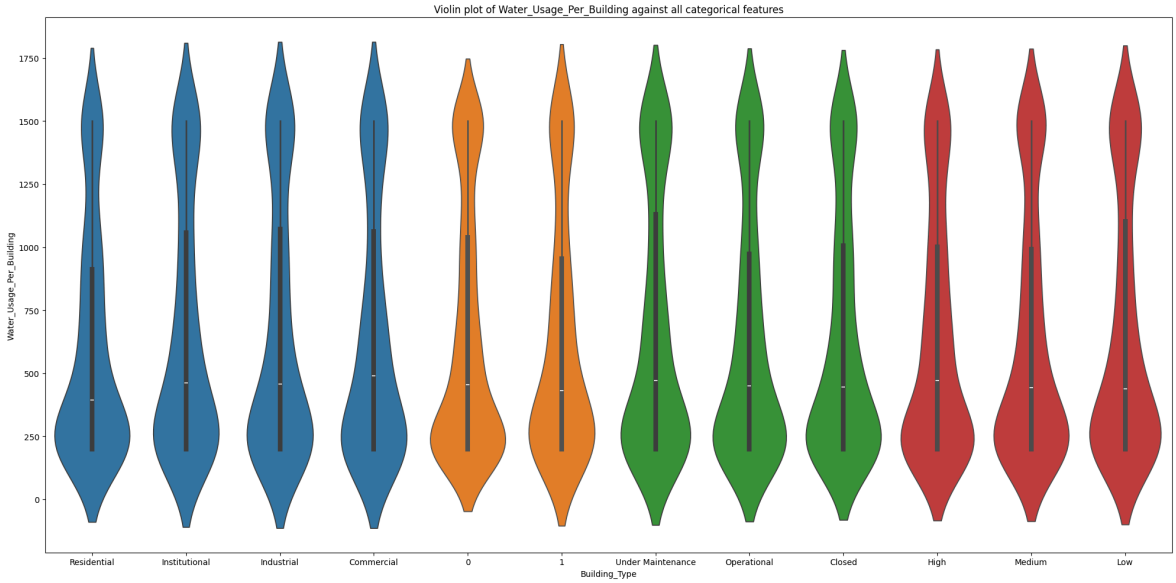
In [6]:
```
# Violin plot
categorical_col = 'Electricity_Bill'

plt.figure(figsize=(20, 10))

for num_column in numeric_df1.columns:
    plt.figure(figsize=(20, 10))
    for cat_column in categorical_df1.columns:
        sns.violinplot(x=cat_column, y=num_column, data=data_frame, orient='v')
    plt.title(f'Violin plot of {num_column} against all categorical features')
    plt.tight_layout()
    plt.show()
```

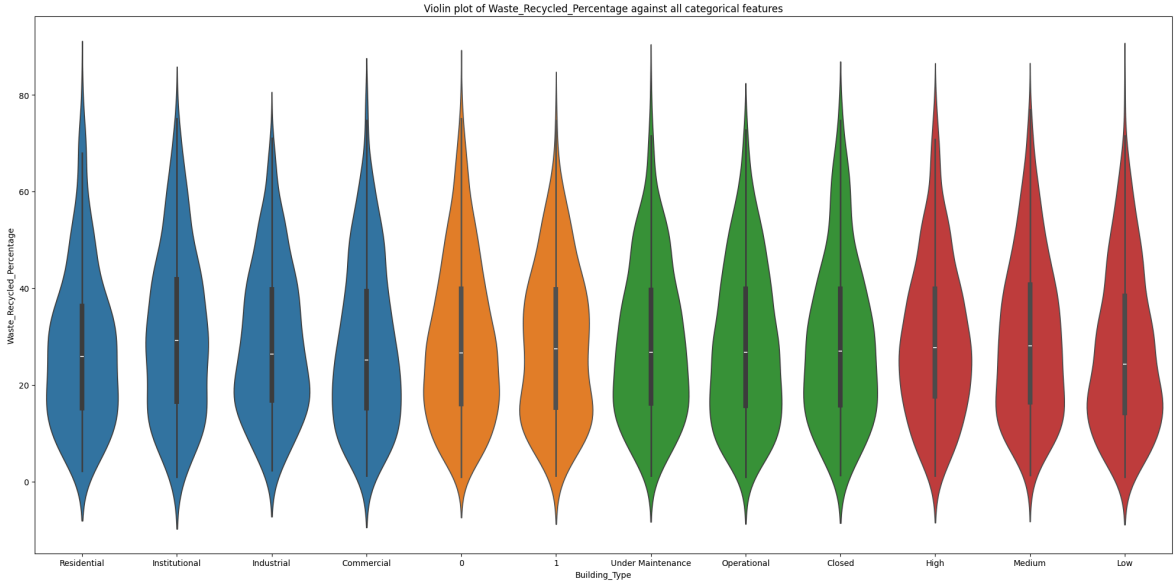
```
<Figure size 2000x1000 with 0 Axes>
```

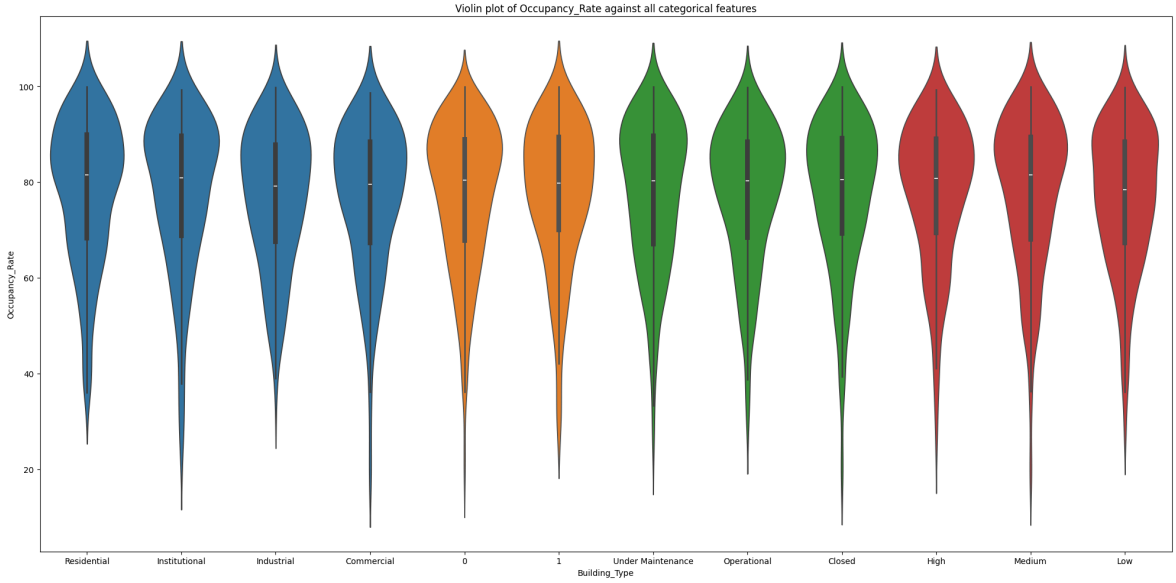


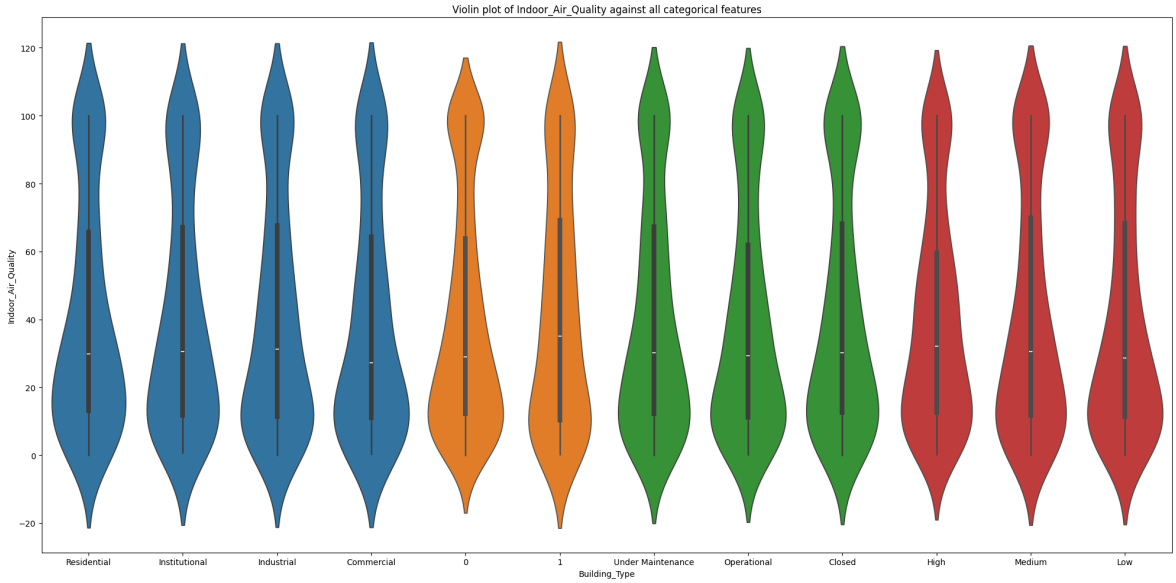


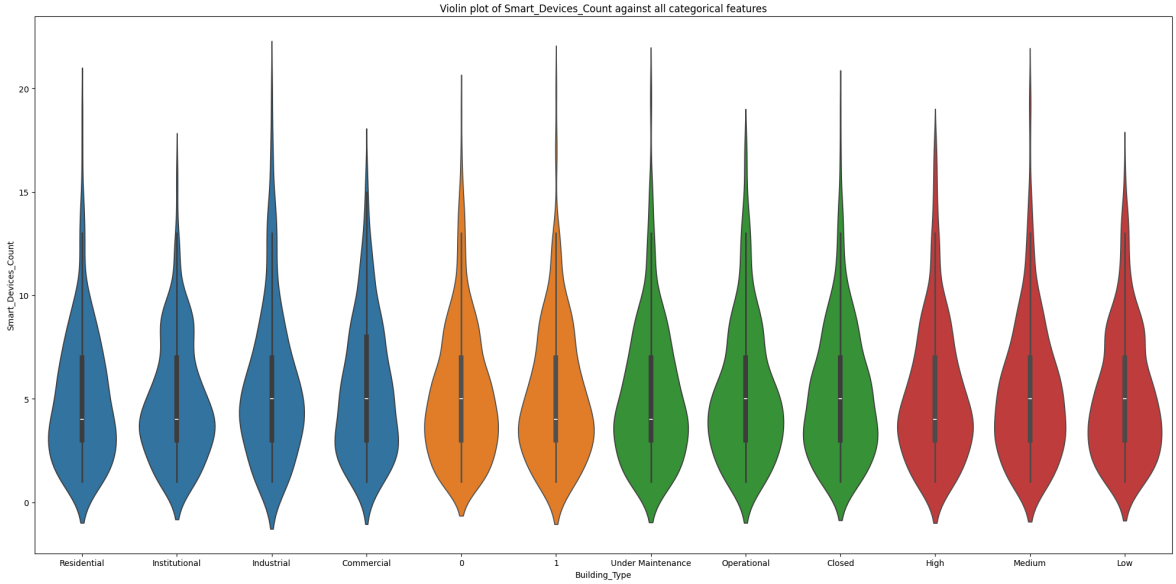

Violin plot of Energy_Consumption_Per_SqM against all categorical features
Energy_Consumption_Per_SqM
Building_Type
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low

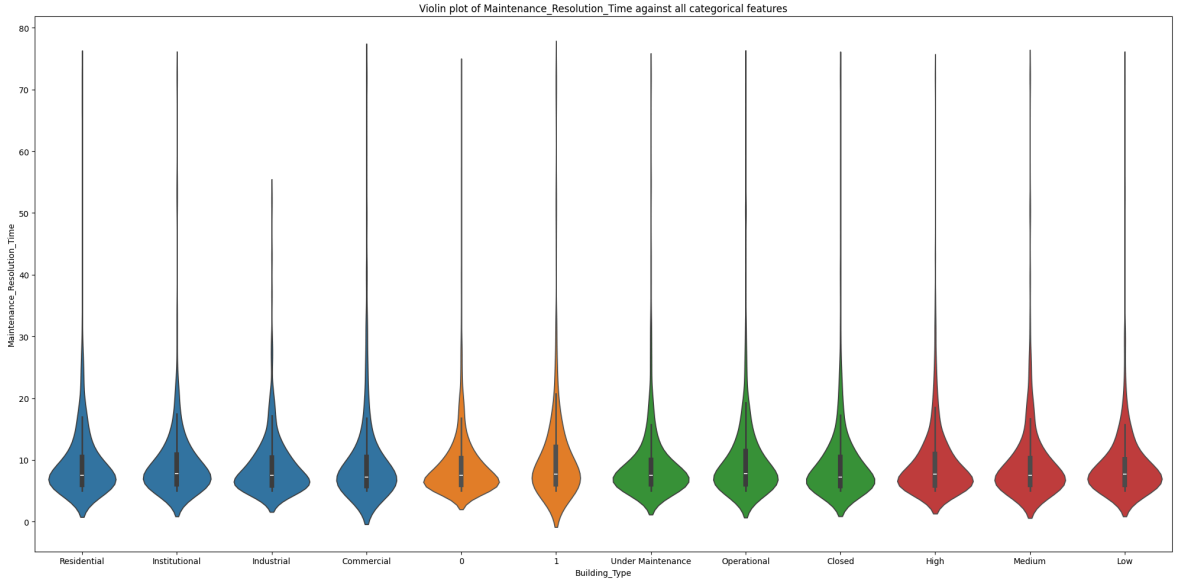

Violin plot of Water_Usage_Per_Building against all categorical features
Water_Usage_Per_Building
Building_Type
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low

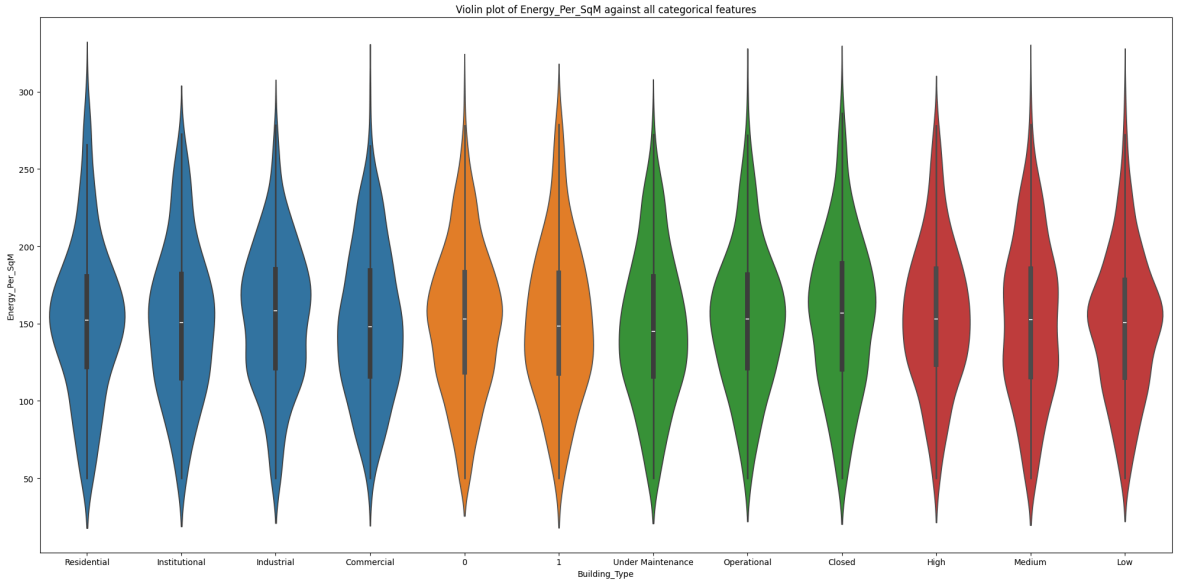

Violin plot of Waste_Recycled_Percentage against all categorical features
Waste_Recycled_Percentage
Building_Type
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low

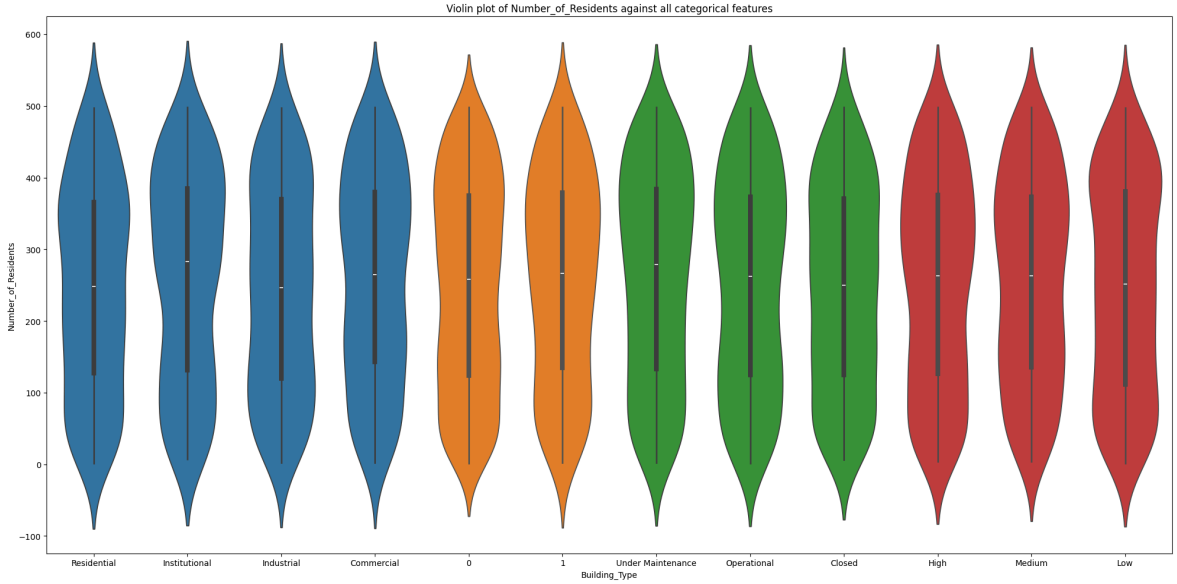

Violin plot of Occupancy_Rate against all categorical features
Occupancy_Rate
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type

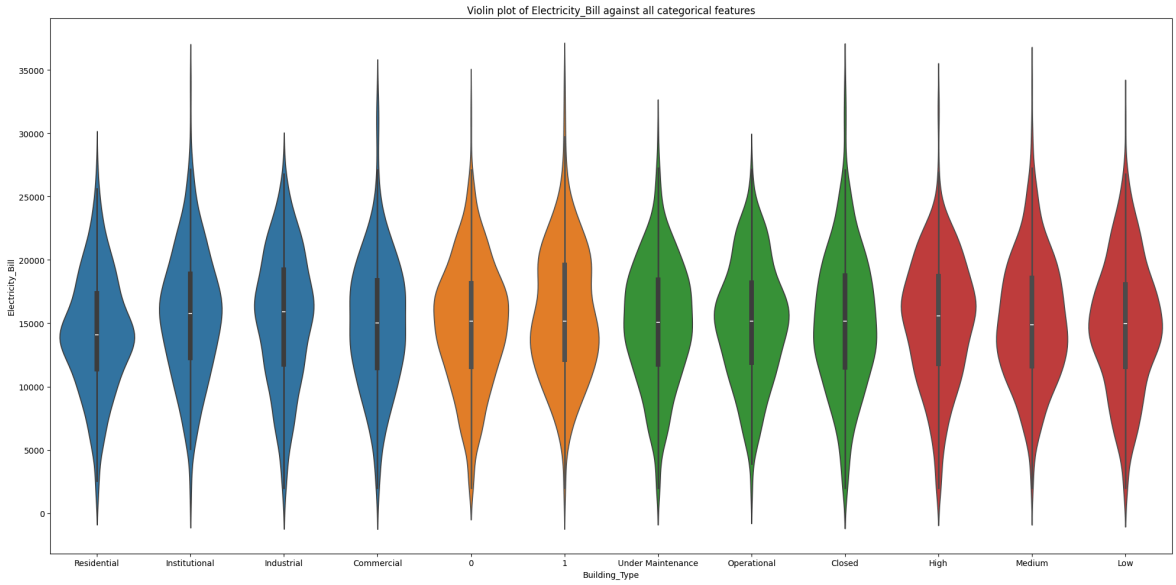

Violin plot of Indoor_Air_Quality against all categorical features
Indoor_Air_Quality
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Smart_Devices_Count against all categorical features
Smart_Devices_Count
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Maintenance_Resolution_Time against all categorical features
Maintenance_Resolution_Time
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Energy_Per_SqM against all categorical features
Energy_Per_SqM
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type


Violin plot of Number_of_Residents against all categorical features
Number_of_Residents
Residential
Institutional
Industrial
Commercial
0
1
Under Maintenance
Operational
Closed
High
Medium
Low
Building_Type



Inferences from the Violinplot of Electricity_Bill against all the categorical features

This violin plot shows the distribution of the electricity bills across different categorical features. The wider sections of the violin plot indicate a higher density of data points, while the narrower sections represent lower density. Here's an inference for each categorical feature based on the plot:

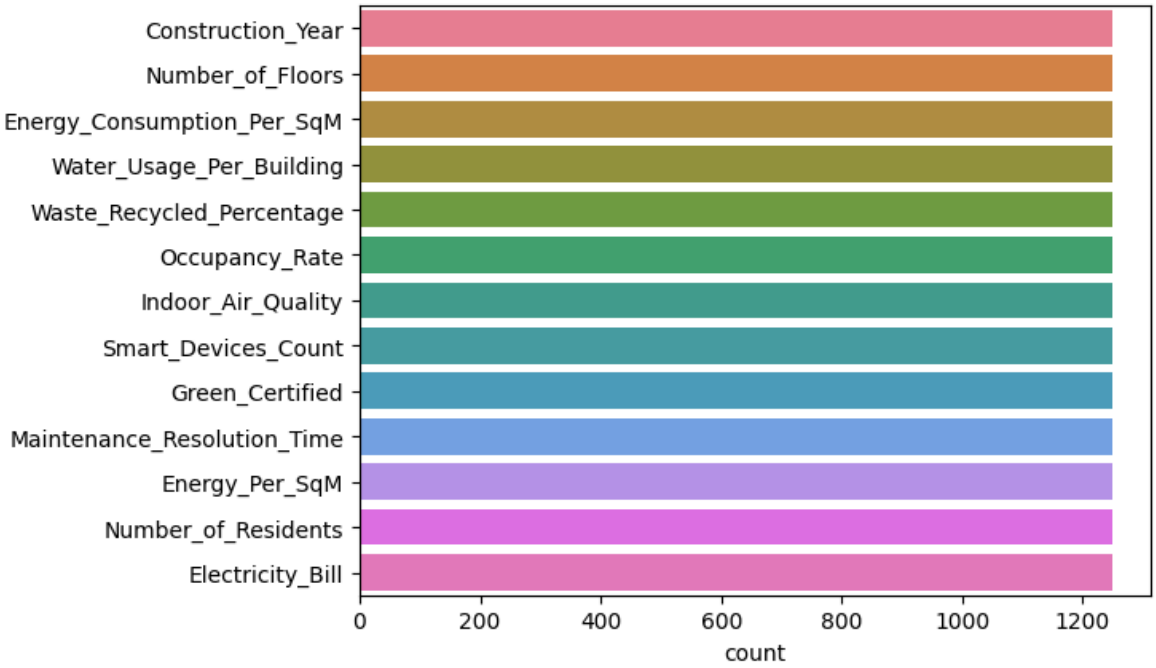
1. Building Type (Residential, Institutional, Industrial, Commercial):
    - Residential and Institutional buildings have a wide spread of electricity bills, with many bills clustering around the 15,000–20,000 range.
    - Industrial and Commercial buildings show a similar spread but appear to have higher upper limits, with some bills reaching over 30,000, though the median values remain consistent across categories. This suggests that industrial and commercial buildings may have more variability in their electricity usage compared to residential and institutional buildings.
2. Green certified (0 and 1):
    - The categories 0 and 1 likely correspond to building being green_certified or not.
    - Both categories show relatively high density around 15,000–20,000, with category 1 possibly having a slightly higher range. This suggests that both categories result in similar electricity usage patterns.
3. Building Condition (Under Maintenance, Operational, Closed):
    - Buildings Under Maintenance have a lower spread of electricity bills, likely due to lower energy usage during maintenance periods.
    - Operational buildings show a higher spread, with bills ranging from near 0 to over 35,000, indicating more active energy usage.
    - Closed buildings have a smaller spread, with most bills clustering around 15,000, but a few outliers suggesting some energy usage even when closed.
4. Energy Efficiency (High, Medium, Low):

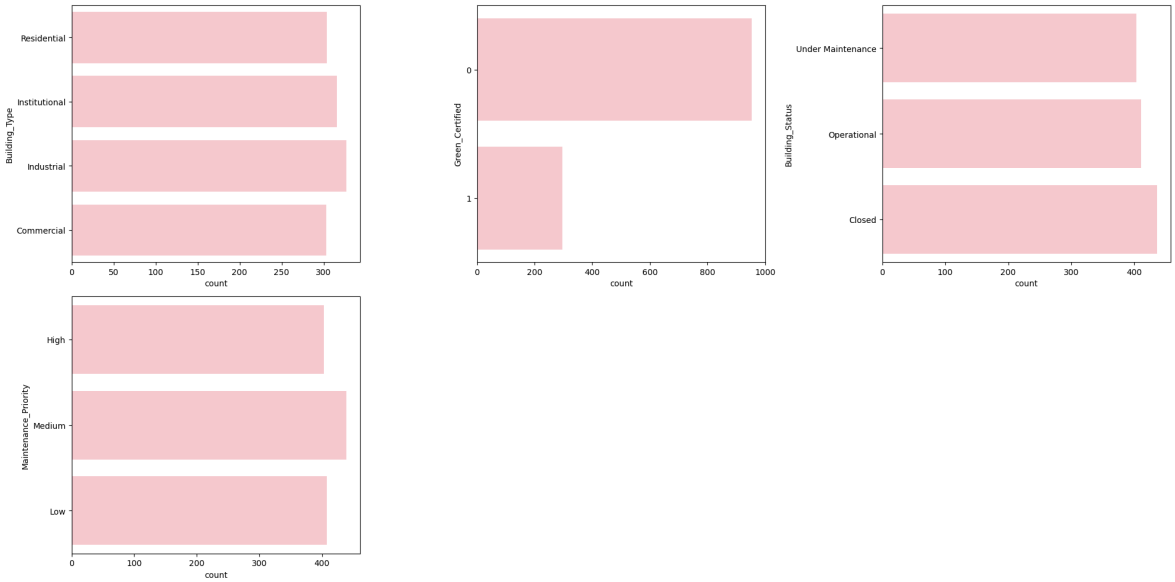
- High-efficiency buildings have a narrower spread, clustering mostly between 10,000 and 20,000, indicating more consistent and likely lower electricity usage.
- Medium and Low-efficiency buildings have a wider spread, especially for Low-efficiency buildings, where there is a greater chance of higher electricity bills, suggesting that lower efficiency correlates with higher energy consumption.

Countplot

- Countplot is a great way to visualize the count of each category in a categorical feature.
- It displays the count of each category in a bar plot.

In [7]:
```
# Count plot
sns.countplot(data_frame, orient='h')

plt.figure(figsize=(20, 10))

for i, column in enumerate(categorical_df1.columns):
    plt.subplot(2, 3, i+1)
    sns.countplot(y=column, data=data_frame_tmp, color= 'pink')

plt.tight_layout()
plt.show()
```





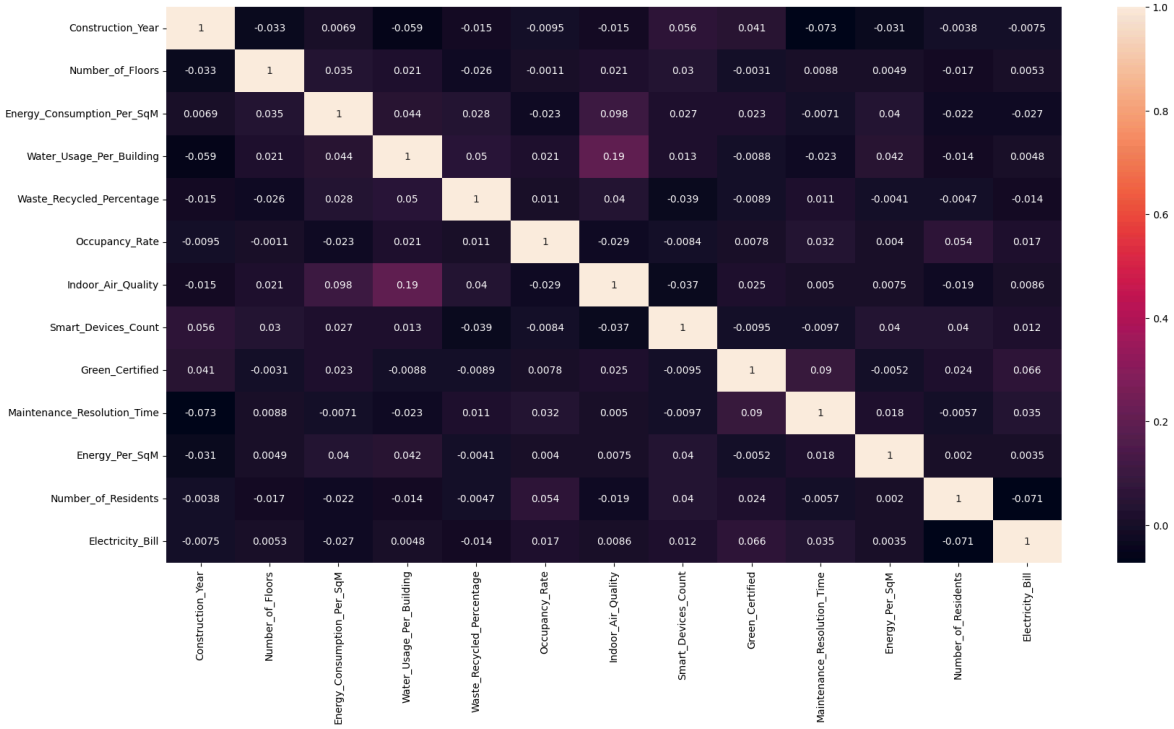
Inferences from the Countplot of the categorical features

1. Only a small fraction of buildings are Green Certified even though such houses are more sustainable.
2. Most buildings are either closed or under maintenance.

Correlation HeatMap

- Correlation heatmap is used to visualize the correlation between all the features in the dataset with colour-coding.
- The correlation coefficient ranges from -1 to 1.
- The closer the correlation coefficient is to 1, the stronger the positive correlation.
- The closer the correlation coefficient is to -1, the stronger the negative correlation.
- The closer the correlation coefficient is to 0, the weaker the correlation.
- The diagonal of the heatmap is always 1 because it represents the correlation of a feature with itself.

In [8]:
```
# Correlation heatmap

plt.figure(figsize=(20, 10))
sns.heatmap(numeric_df.corr(), annot=True)
plt.show()
```



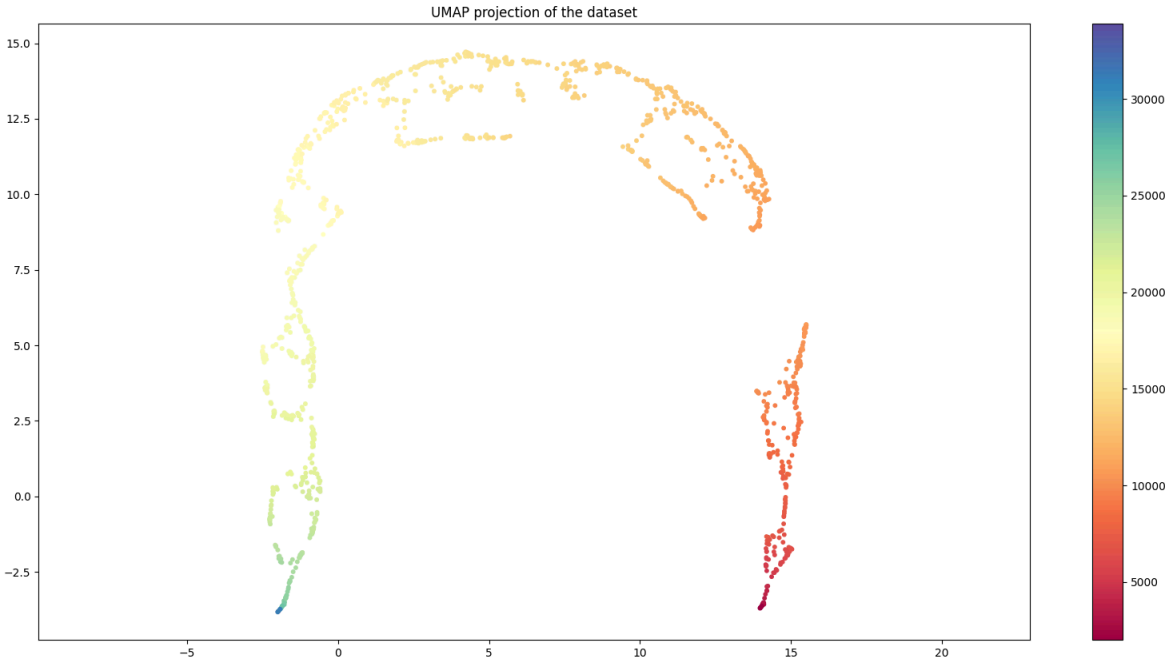
Inferences from the Correlation HeatMap

1. Energy_Consumption_per_SqM and Water_Usage_per_Building both have a strong positive correlation with Indoor_Air_Quality.
2. Maintenance_Resolution_Time and Electricity_Bill both have a strong positive correlation with Green_certified buildings.
3. Energy_Consumption_per_SqM has a strong negative correlation with Occupancy_Rate.
4. Constructio_year and Water_Usage_per_Building have a strong negative correlation.
5. Number_of_Residents and Electricity_Bill have a strong negative correlation.

(b) UMAP

UMAP is a dimensionality reduction technique that is used to visualize high-dimensional data in a lower-dimensional space. It is similar to t-SNE but is faster and scales better to large datasets. UMAP is used to visualize the data in 2D space.

In [9]:
```
# Use the Uniform Manifold Approximation and Projection (UMAP) algorithm to redu
reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(numeric_df)
plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = numeric_df['Electricity_Bill']
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with Electricity_Bill labels')

plt.show()
```



Inferences from the UMAP plot of unscaled data

- We see a large number of small clusters in the UMAP plot.
- The data points show that they are not linearly separable, hence the following linear regression methods will show poor performance on this dataset.
- Some clusters are dense and some are sparse, indicating that the data is not uniformly distributed.

(c) Preprocessing data and applying Linear Regression

Min-Max Scaling for numerical features and label encoding for categorical features

In [10]:
```
# Perform the necessary pre-processing steps, including handling missing values

# data_frame.dropna(inplace=True)

scaler = MinMaxScaler()
encoder = LabelEncoder()

X = data_frame.drop(columns=['Electricity_Bill'])
y = data_frame['Electricity_Bill']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_

X_train_num = X_train.select_dtypes(include=[np.number])
X_train_cat = X_train.drop(columns=X_train_num.columns)

X_test_num = X_test.select_dtypes(include=[np.number])
X_test_cat = X_test.drop(columns=X_test_num.columns)
```

```
X_train_scaled_num = pd.DataFrame(scaler.fit_transform(X_train_num), columns=X_t

# X_train_scaled_cat = X_train_cat.apply(lambda col: encoder.fit_transform(col.a

X_cat_combined = pd.concat([X_train_cat, X_test_cat])

X_cat_combined_encoded = X_cat_combined.apply(lambda col: encoder.fit_transform(

X_train_scaled_cat = X_cat_combined_encoded.loc[X_train_cat.index]
X_test_scaled_cat = X_cat_combined_encoded.loc[X_test_cat.index]


X_train_scaled_num.reset_index(drop=True, inplace=True)
X_train_scaled_cat.reset_index(drop=True, inplace=True)


X_train_scaled = pd.concat([X_train_scaled_num, X_train_scaled_cat], axis=1)


# print(X_test_num)
# print(X_test_cat)

X_test_scaled_num = pd.DataFrame(scaler.transform(X_test_num), columns=X_test_nu

# X_test_scaled_cat = X_test_cat.apply(lambda col: encoder.transform(col.astype(

X_test_scaled_num.reset_index(drop=True, inplace=True)
X_test_scaled_cat.reset_index(drop=True, inplace=True)

X_test_scaled = pd.concat([X_test_scaled_num, X_test_scaled_cat], axis=1)

data_frame_scaled = pd.concat([X_train_scaled, y_train], axis=1)
```

In [11]:
```
X_train
```

Out[11]:

	Building_Type	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM
1194	Residential	1994	12	50.00000
911	Institutional	1984	13	50.00000
422	Institutional	2005	4	50.00000
670	Residential	1974	1	50.00000
931	Commercial	1984	11	250.00000
...	...	...	...	.
1044	Commercial	2017	13	250.00000
1095	Commercial	2013	6	50.00000
1130	Commercial	1986	11	192.78833
860	Industrial	2015	2	224.10038
1126	Commercial	2011	3	50.00000

1000 rows × 15 columns

In [12]:
```
X_train_scaled_num
```

Out[12]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 12 columns

In [13]:
```
X_train_scaled_cat
```

Out[13]:

	Building_Type	Building_Status	Maintenance_Priority
0	3	1	1
1	2	1	1
2	2	2	1
3	3	2	2
4	0	2	0
...	...	...	...
995	0	1	2
996	0	1	1
997	0	1	0
998	1	0	0
999	0	1	2

1000 rows × 3 columns

In [14]:
```
X_train_scaled
```

Out[14]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 15 columns

In [15]:
```
y_train
```

Out[15]:
```
1194    17591.376140
911     13847.735490
422     15398.443910
670     13197.166340
931     16815.610080
           ...
1044    14828.987040
1095     5191.145924
1130    23362.030160
860      9398.009440
1126    10990.719530
Name: Electricity_Bill, Length: 1000, dtype: float64
```

In [16]:
```
X_test
```

Out[16]:

	Building_Type	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM
680	Residential	1991	1	250.00000
1102	Institutional	1998	10	250.00000
394	Industrial	2007	10	50.00000
930	Industrial	1996	6	50.00000
497	Commercial	2018	8	50.00000
...	...	...	...	.
382	Industrial	1977	11	74.35493
678	Industrial	1970	9	250.00000
1002	Residential	1976	13	250.00000
361	Institutional	2006	11	77.31203
490	Institutional	2004	12	50.00000

250 rows × 15 columns

In [17]:
```
X_test_scaled
```

Out[17]:

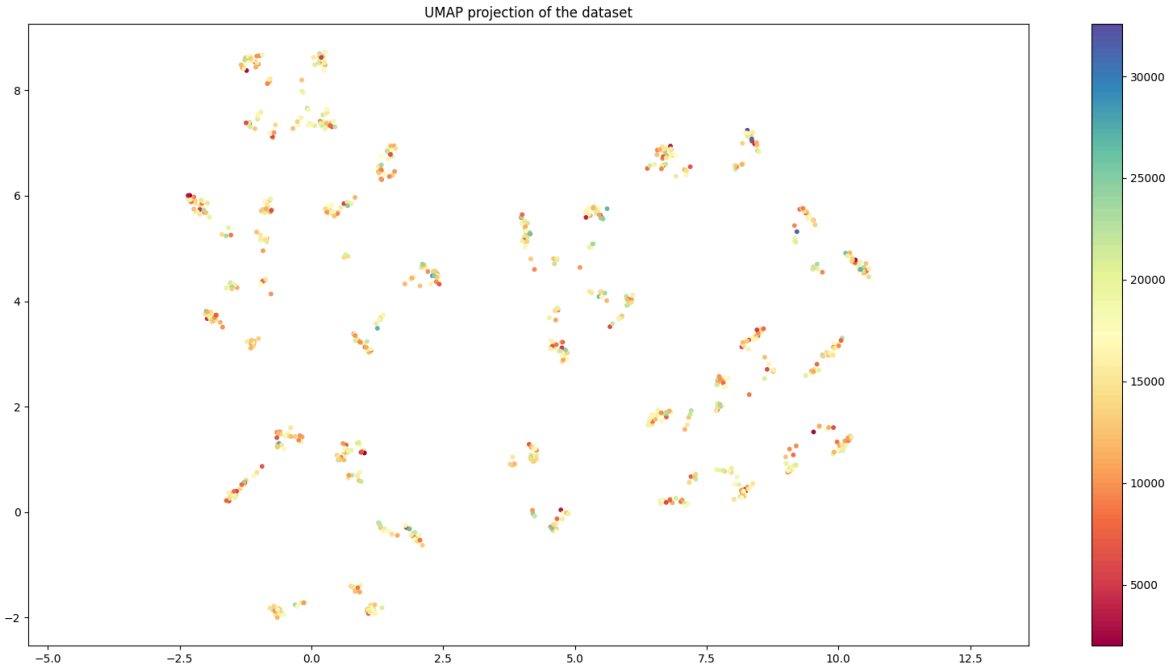
	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.396226	0.000000	1.000000	
1	0.528302	0.750000	1.000000	
2	0.698113	0.750000	0.000000	
3	0.490566	0.416667	0.000000	
4	0.905660	0.583333	0.000000	
...	...	...	...	
245	0.132075	0.833333	0.121775	
246	0.000000	0.666667	1.000000	
247	0.113208	1.000000	1.000000	
248	0.679245	0.833333	0.136560	
249	0.641509	0.916667	0.000000	

250 rows × 15 columns

In [18]:
```
y_test
```

Out[18]:
```
680      8453.948237
1102    11432.649750
394     11702.042480
930     14251.068910
497      9785.154833
            ...
382     17541.382160
678     16243.691150
1002    11987.080930
361     11816.911900
490     25433.128080
Name: Electricity_Bill, Length: 250, dtype: float64
```

In [19]:
```
reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_scaled)
plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset')

plt.show()
```



Inferences from the UMAP of normalized data

- The data is divided into a large number of very small clusters.
- The data is not linearly separable however there is a more noticeable separation between the clusters.
- The data is more uniformly distributed than the unscaled data.

Linear Regression

In [20]:
```
# Applying linear regression model on pre-processed data

N = len(y_test)
p = X_test.shape[1]

N_train = len(y_train)
p_train = X_train.shape[1]

model = LinearRegression()
model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)
y_pred_train = model.predict(X_train_scaled)

R2 = model.score(X_test_scaled, y_test)
R2_train = model.score(X_train_scaled, y_train)

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train - 1)

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train))
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', model.score(X_train_scaled, y_train))
```

```
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train)))
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n')
```

```
Mean Squared Error for train: 24475013.16847547
Mean Squared Error for test: 24278016.155742623

Root Mean Squared Error for train: 4947.222773281538
Root Mean Squared Error for test: 4927.272689403604

R2 Score for train: 0.013922520844610209
R2 Score for test: 3.7344733075372893e-05

Adjusted R2 Score for train: -0.0011091480449536562
Adjusted R2 Score for test: -0.0640628254763429

Mean Absolute Error for train: 4006.3284693293604
Mean Absolute Error for test: 3842.409312558516
```

Inferences from the Linear Regression

- The large mean squared error and root mean squared error are not only because the model is unfit for our data but also because we have not scaled the label and hence the error ends up being large.
- The R2 score of 3.7344733075372893e-05 indicates that the model is not a good fit for the data. and is performing poorly.

(d) Recursive Feature Elimination (RFE) and Correlation Analysis

RFE

In [22]:
```
# Perform Recursive Feature Elimination (RFE) or Correlation analysis on the ori

rfe = RFE(model, n_features_to_select=3)

rfe.fit(X_train_scaled, y_train)

selected_features = X_train_scaled.columns[rfe.support_]
print(f"Selected Features by RFE: {selected_features}", '\n')

X_train_rfe = X_train_scaled[selected_features]
X_test_rfe = X_test_scaled[selected_features]

model.fit(X_train_rfe, y_train)

y_pred_rfe = model.predict(X_test_rfe)
y_pred_train_rfe = model.predict(X_train_rfe)
```

```
R2_rfe = model.score(X_test_rfe, y_test)
R2_train_rfe = model.score(X_train_rfe, y_train)

N = len(y_test)
p = X_test_rfe.shape[1]

N_train = len(y_train)
p_train = X_train_rfe.shape[1]

Adjusted_R2_rfe = 1 - (1 - R2_rfe) * (N - 1) / (N - p - 1)
Adjusted_R2_train_rfe = 1 - (1 - R2_train_rfe) * (N_train - 1) / (N_train - p_tr

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train_
print('Mean Squared Error:', mean_squared_error(y_test, y_pred_rfe), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error:", np.sqrt(mean_squared_error(y_test, y_pred_rfe)

print('R2 Score for train:', model.score(X_train_rfe, y_train))
print('R2 Score:', model.score(X_test_rfe, y_test), '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train_rfe)
print("Adjusted R2 Score:", Adjusted_R2_rfe, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train_rf
print("Mean Absolute Error:", np.mean(np.abs(y_test - y_pred_rfe)), '\n')
```

```
Selected Features by RFE: Index(['Green_Certified', 'Maintenance_Resolution_Tim
e',
       'Number_of_Residents'],
      dtype='object')

Mean Squared Error for train: 24598921.04560434
Mean Squared Error: 23976300.350124482

Root Mean Squared Error for train: 4959.729936761108
Root Mean Squared Error: 4896.560052743607

R2 Score for train: 0.008930377784842514
R2 Score: 0.012464411927795571

Adjusted R2 Score for train: 0.005945228320339058
Adjusted R2 Score: 0.00042129500008580845

Mean Absolute Error for train: 4017.1253534034668
Mean Absolute Error: 3816.7223458373137
```

Inferences from the RFE

- The MSE has decreased by almost 301716 units and the R2 score has increased significantly. This indicates that RFE was successful in the selecting the three most important features for the model.

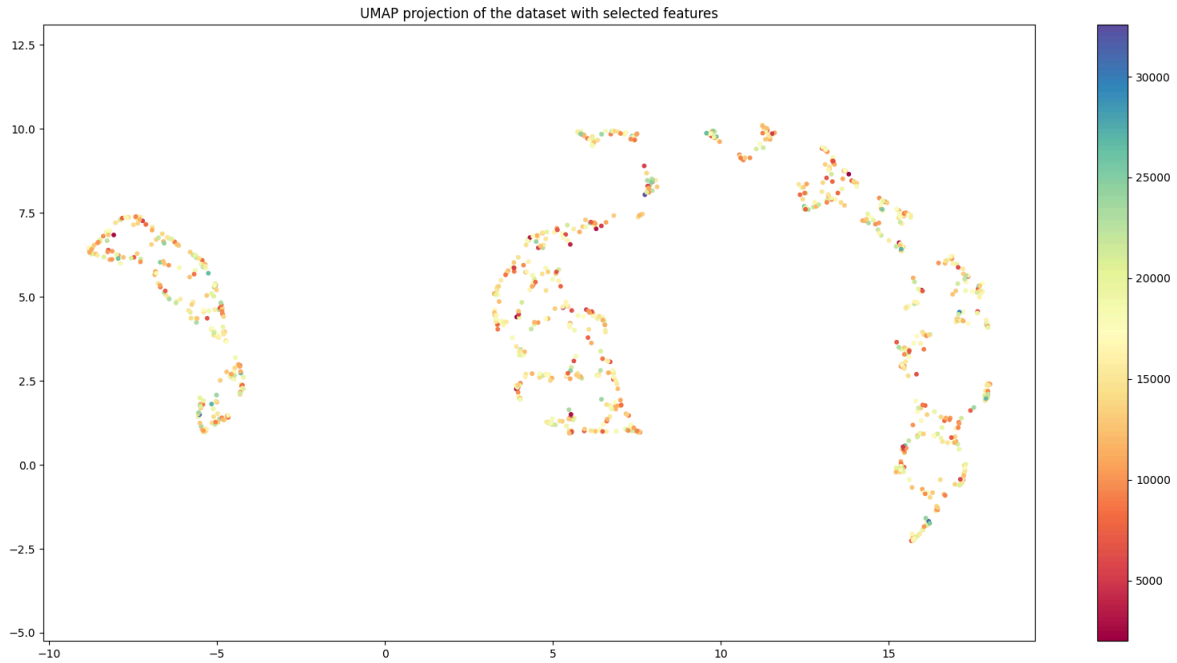
In [46]:
```
X_train_rfe
```

Out[46]:

	Green_Certified	Maintenance_Resolution_Time	Number_of_Residents
0	0.0	0.009899	0.931590
1	0.0	0.011183	0.162978
2	1.0	0.032088	0.659960
3	1.0	0.098298	0.768612
4	1.0	0.029140	0.074447
...	...	...	...
995	0.0	0.052232	0.096579
996	0.0	0.189633	0.653924
997	1.0	1.000000	0.309859
998	0.0	0.045982	0.140845
999	1.0	0.024654	0.943662

1000 rows × 3 columns

In [47]:
```
# UMAP projection of the dataset with selected features

reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_rfe)

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with selected features')

plt.show()
```



The UMAP plot shows distribution of data in approximately three clusters due to our feature selection using RFE.

Correlation Analysis

In [24]:
```
corr_matrix = data_frame_scaled.corr()

corr_matrix['Electricity_Bill'].sort_values(ascending=False)

selected_features_corr = corr_matrix['Electricity_Bill'].sort_values(ascending=F

X_train_corr = X_train_scaled[selected_features_corr]
X_test_corr = X_test_scaled[selected_features_corr]

model.fit(X_train_corr, y_train)

y_pred_corr = model.predict(X_test_corr)
y_pred_train_corr = model.predict(X_train_corr)

R2_corr = model.score(X_test_corr, y_test)
R2_train_corr = model.score(X_train_corr, y_train)

N = len(y_test)
p = X_test_corr.shape[1]

N_train = len(y_train)
p_train = X_train_corr.shape[1]

Adjusted_R2_corr = 1 - (1 - R2_corr) * (N - 1) / (N - p - 1)
Adjusted_R2_train_corr = 1 - (1 - R2_train_corr) * (N_train - 1) / (N_train - p_

print("Features selected by correlation analysis: ", selected_features_corr, '\n

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train_
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred_corr), '

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', model.score(X_train_corr, y_train))
print('R2 Score for test:', model.score(X_test_corr, y_test), '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train_corr)
print("Adjusted R2 Score for test:", Adjusted_R2_corr, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train_co
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred_corr)), '\
```

```
Features selected by correlation analysis:  Index(['Number_of_Residents', 'Occupa
ncy_Rate', 'Building_Status'], dtype='object')

Mean Squared Error for train: 24695588.591842424
Mean Squared Error for test: 24218279.313990254

Root Mean Squared Error for train: 4969.46562437476
Root Mean Squared Error for test: 4921.207099278617

R2 Score for train: 0.005035724505008332
R2 Score for test: 0.0024977850966028114

Adjusted R2 Score for train: 0.002038844157131847
Adjusted R2 Score for test: -0.00966687606075567

Mean Absolute Error for train: 4011.4533029079444
Mean Absolute Error for test: 3823.7875391944817
```

Inferences from the Correlation Analysis

Similar to RFE, the correlation analysis method also improved the model performance, although not as significantly as RFE, indicating RFE was more effective in selecting the most important features.

In [49]:
```
X_train_corr
```

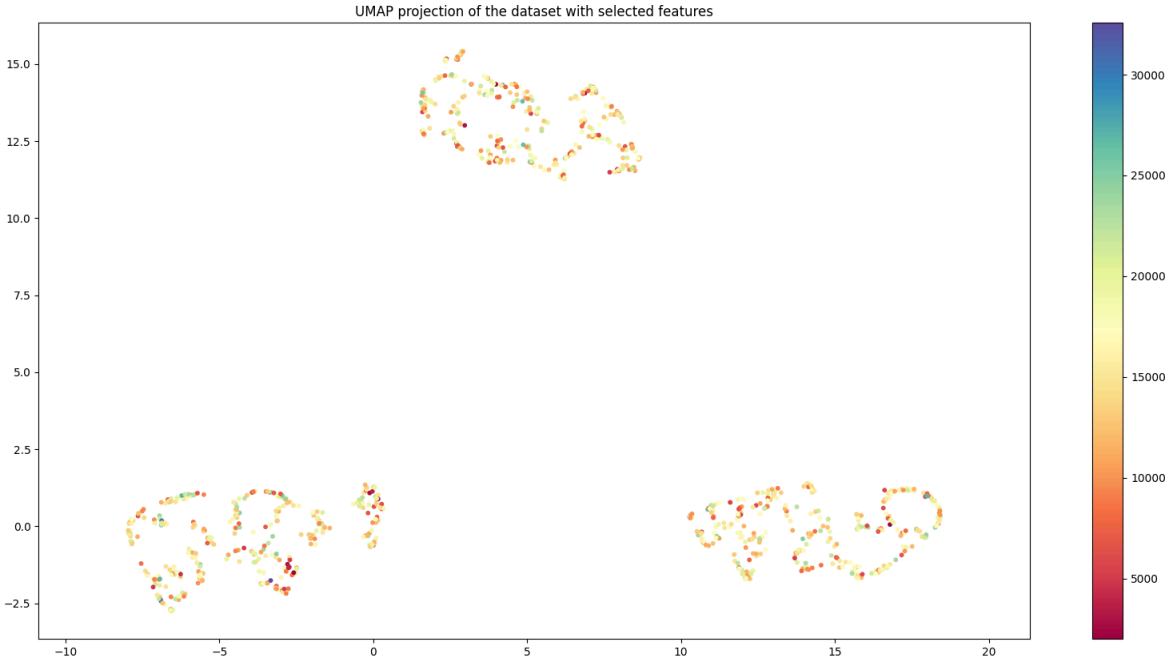
Out[49]:

	Number_of_Residents	Occupancy_Rate	Building_Status
0	0.931590	0.761137	1
1	0.162978	0.521143	1
2	0.659960	0.881262	2
3	0.768612	0.686595	2
4	0.074447	0.508822	2
...	...	...	...
995	0.096579	0.953991	1
996	0.653924	0.540006	1
997	0.309859	0.390989	1
998	0.140845	0.951936	0
999	0.943662	0.715220	1

1000 rows × 3 columns

In [48]:
```
# UMAP projection of the dataset with selected features

reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_corr)

plt.figure(figsize=(20, 10))
```

```
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with selected features')

plt.show()
```



The UMAP shows a huge improvement in the separability of the three clusters after feature selection using correlation analysis.

(e) One Hot Encoding and applying Ridge Regression

One Hot Encoding

In [26]:
```
# One-hot encoding categorical features
X_train_cat_encoded = pd.get_dummies(X_train_cat).reset_index(drop=True)
X_test_cat_encoded = pd.get_dummies(X_test_cat).reset_index(drop=True)

# Ensure that both train and test sets have the same columns after one-hot encod
X_train_cat_encoded, X_test_cat_encoded = X_train_cat_encoded.align(X_test_cat_e

# Concatenate the scaled numerical features with the one-hot encoded categorical
X_train_scaled_hot = pd.concat([X_train_scaled_num, X_train_cat_encoded], axis=1
X_test_scaled_hot = pd.concat([X_test_scaled_num, X_test_cat_encoded], axis=1)
```

In [28]:
```
X_train_cat_encoded
```

Out[28]:

	Building_Type_Commercial	Building_Type_Industrial	Building_Type_Institutional	Bu
0	False	False	False	
1	False	False	True	
2	False	False	True	
3	False	False	False	
4	True	False	False	
...	...	...	...	
995	True	False	False	
996	True	False	False	
997	True	False	False	
998	False	True	False	
999	True	False	False	

1000 rows × 10 columns

In [29]:
```
X_train_scaled_hot
```

Out[29]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 22 columns

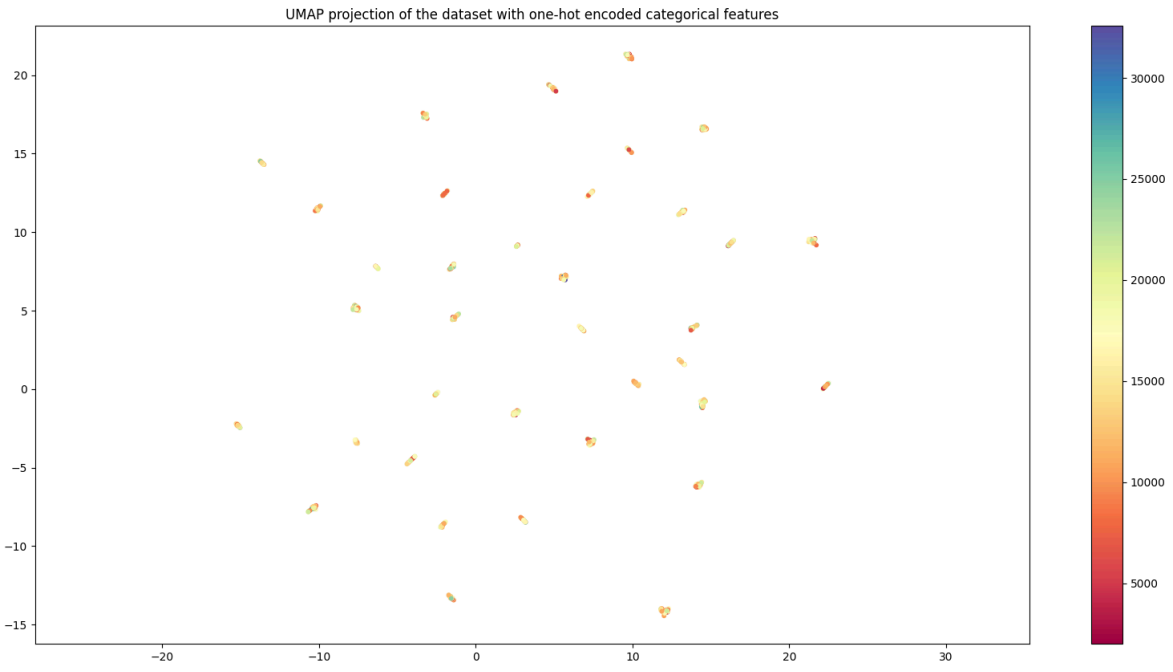
In [30]:
```
X_test_scaled_hot
```

Out[30]:

	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.396226	0.000000	1.000000	
1	0.528302	0.750000	1.000000	
2	0.698113	0.750000	0.000000	
3	0.490566	0.416667	0.000000	
4	0.905660	0.583333	0.000000	
...	...	...	...	
245	0.132075	0.833333	0.121775	
246	0.000000	0.666667	1.000000	
247	0.113208	1.000000	1.000000	
248	0.679245	0.833333	0.136560	
249	0.641509	0.916667	0.000000	

250 rows × 22 columns

In [50]:
```
# UMAP projection of the dataset with one-hot encoded categorical features

reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_scaled_hot)

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral', s
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with one-hot encoded categorical featu

plt.show()
```



The UMAP plot shows clear distinction between various small clusters indicating that one hot encoding has improved the separability of the data.

Ridge Regression

In [31]:
```
# Applying ridge regression model on pre-processed data

model = Ridge(alpha=0.05)

model.fit(X_train_scaled_hot, y_train)

y_pred = model.predict(X_test_scaled_hot)
y_pred_train = model.predict(X_train_scaled_hot)

R2 = model.score(X_test_scaled_hot, y_test)
R2_train = model.score(X_train_scaled_hot, y_train)

N = len(y_test)
p = X_test_scaled_hot.shape[1]

N_train = len(y_train)
p_train = X_train_scaled_hot.shape[1]

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train - 1)

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train))
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', model.score(X_train_scaled_hot, y_train))
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train)))
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n')
```

```
Mean Squared Error for train: 24188925.944940828
Mean Squared Error for test: 24129699.306866154

Root Mean Squared Error for train: 4918.223860799834
Root Mean Squared Error for test: 4912.199029647125

R2 Score for train: 0.0254487319342519
R2 Score for test: 0.006146217429753964

Adjusted R2 Score for train: 0.003503872264398855
Adjusted R2 Score for test: -0.09017441348013766

Mean Absolute Error for train: 3976.6864225405034
Mean Absolute Error for test: 3797.4489513870913
```

Inferences from the Ridge Regression on the One Hot Encoded data

- R2 score has seen a huge jump from 0.000037 which we obtaine dby applying linear regression on the unscaled data to 0.006.

These predictions indicate that ridge regression has done a better job at predicting the target value than logistic regression. These values are also better than regression after appyling recursive feature elimination. This implies that Ridge Regression has implemented regularization, but its effectiveness in improving model performance is limited as RMSE, MSE have improved but still the values as in are large and low R^2 score although it has a large jump. This again leads us to conclude that the model is not a good fit for the data as data is not linearly separable and needs a more complex model to predict the target value.

In [32]:
```
X_train_scaled
```

Out[32]:

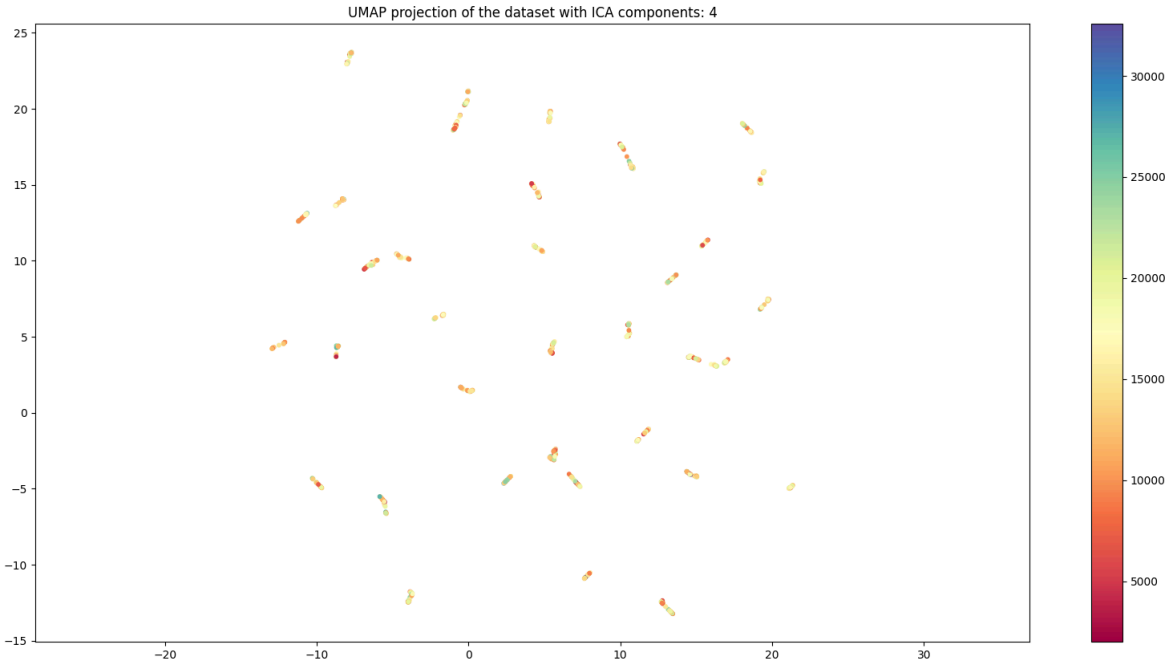
	Construction_Year	Number_of_Floors	Energy_Consumption_Per_SqM	Water_Usage_I
0	0.452830	0.916667	0.000000	
1	0.264151	1.000000	0.000000	
2	0.660377	0.250000	0.000000	
3	0.075472	0.000000	0.000000	
4	0.264151	0.833333	1.000000	
...	...	...	...	
995	0.886792	1.000000	1.000000	
996	0.811321	0.416667	0.000000	
997	0.301887	0.833333	0.713942	
998	0.849057	0.083333	0.870502	
999	0.773585	0.166667	0.000000	

1000 rows × 15 columns

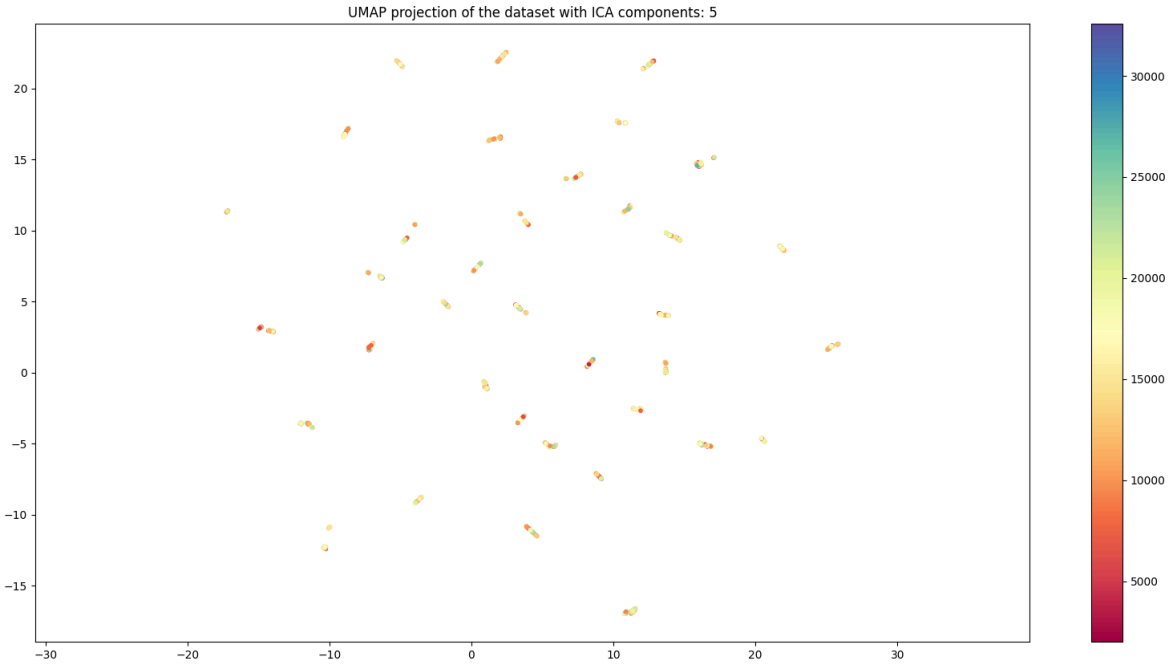
(f) Independent Component Analysis (ICA)

In [51]:
```
# Independent Component Analysis (ICA)
from sklearn.decomposition import FastICA

components = [4, 5, 6, 8]

for component in components:
    ica = FastICA(n_components=component)
    X_train_ica = ica.fit_transform(X_train_scaled_hot)
```

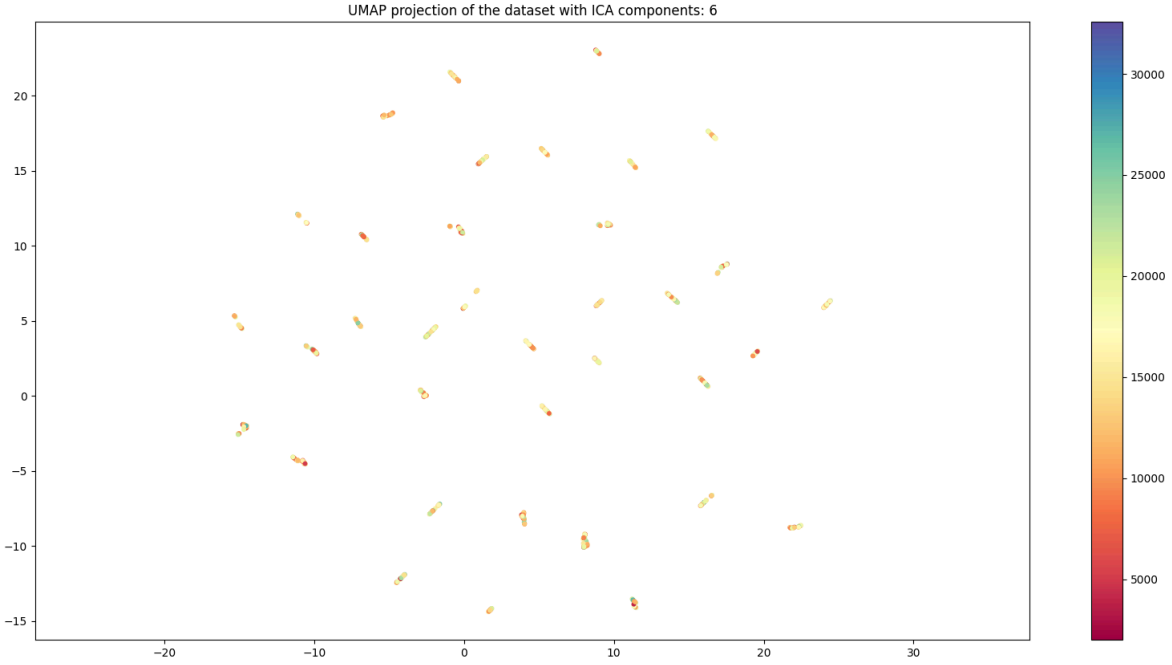
```
# UMAP projection of the dataset with ICA components
reducer = umap.UMAP(n_components=2)
embedding = reducer.fit_transform(X_train_ica)

plt.figure(figsize=(20, 10))
plt.scatter(embedding[:, 0], embedding[:, 1], c = y_train, cmap = 'Spectral'
plt.gca().set_aspect('equal', 'datalim')
plt.colorbar()
plt.title('UMAP projection of the dataset with ICA components: ' + str(compo

plt.show()

X_test_ica = ica.transform(X_test_scaled_hot)

model = LinearRegression()

model.fit(X_train_ica, y_train)

y_pred = model.predict(X_test_ica)
y_pred_train = model.predict(X_train_ica)

R2 = model.score(X_test_ica, y_test)
R2_train = model.score(X_train_ica, y_train)

N = len(y_test)
p = X_test_ica.shape[1]

N_train = len(y_train)
p_train = X_train_ica.shape[1]

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train

print("number of components: ", component, '\n')

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_tr
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_tra
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test

print('R2 Score for train:', model.score(X_train_ica, y_train))
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_trai
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n
```



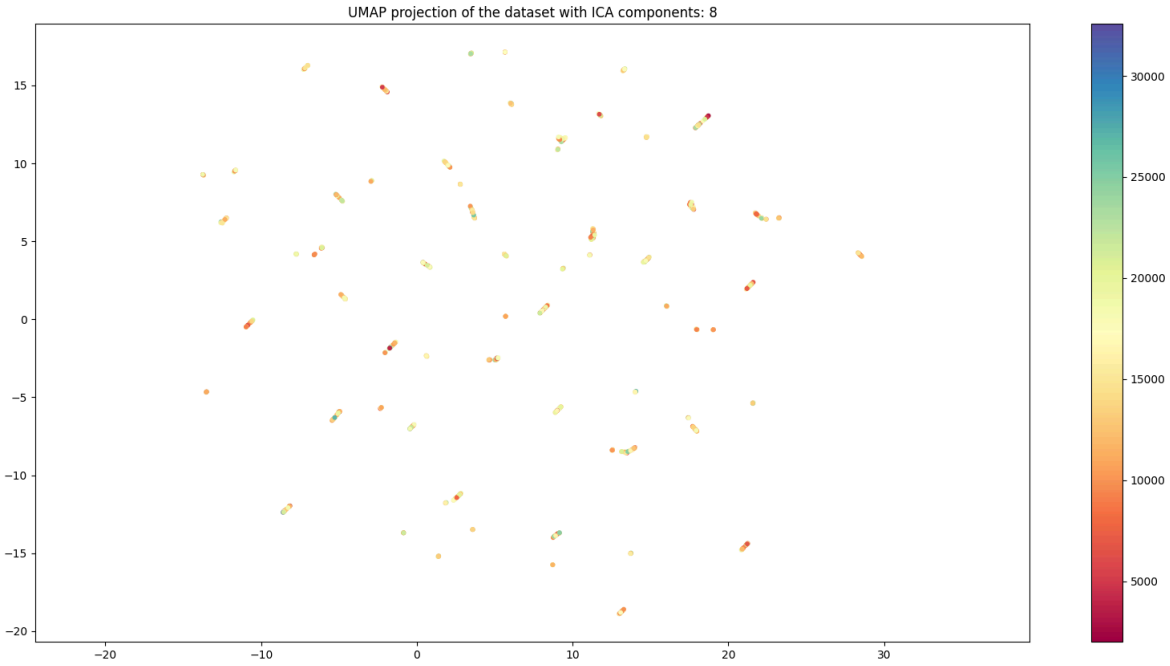

```
number of components:  4

Mean Squared Error for train: 24723510.292957045
Mean Squared Error for test: 24639889.43607532

Root Mean Squared Error for train: 4972.274157059026
Root Mean Squared Error for test: 4963.858321515162

R2 Score for train: 0.003910782897855292
R2 Score for test: -0.014867487850868777

Adjusted R2 Score for train: -9.359586436441525e-05
Adjusted R2 Score for test: -0.031436752958638126

Mean Absolute Error for train: 3999.4510999309614
Mean Absolute Error for test: 3854.3454182518394
```



```
number of components:  5

Mean Squared Error for train: 24680440.647721324
Mean Squared Error for test: 24698036.226428285

Root Mean Squared Error for train: 4967.941288674951
Root Mean Squared Error for test: 4969.711885655775

R2 Score for train: 0.005646022299354736
R2 Score for test: -0.017262437195312952

Adjusted R2 Score for train: 0.0006442417274198364
Adjusted R2 Score for test: -0.0381079789411185

Mean Absolute Error for train: 3993.727088053057
Mean Absolute Error for test: 3839.755733501596
```




```
number of components:  6

Mean Squared Error for train: 24616634.383210227
Mean Squared Error for test: 24493735.346131023

Root Mean Squared Error for train: 4961.5153313489045
Root Mean Squared Error for test: 4949.114602242609

R2 Score for train: 0.008216722467331072
R2 Score for test: -0.008847694844665854

Adjusted R2 Score for train: 0.0022240742647167133
Adjusted R2 Score for test: -0.03375751447046005

Mean Absolute Error for train: 3986.3770717768302
Mean Absolute Error for test: 3823.865663480825
```




```
number of components:  8

Mean Squared Error for train: 24414626.128019493
Mean Squared Error for test: 24074349.40562409

Root Mean Squared Error for train: 4941.115878829345
Root Mean Squared Error for test: 4906.561872189537

R2 Score for train: 0.0163554633814168
R2 Score for test: 0.008425968537081663

Adjusted R2 Score for train: 0.008414841491458547
Adjusted R2 Score for test: -0.024489352009405163

Mean Absolute Error for train: 3978.83358502663
Mean Absolute Error for test: 3781.746734661708
```

Independent Component Analysis (ICA) is a dimensionality reduction technique that is used to separate a multivariate signal into additive, independent components. It is used to separate the data into independent components.

Inferences from applying ICA

- The UMAP plot shows a clear separation between the clusters, indicating that ICA has done a good job in separating the data into independent components.
- We see that R2 score decreases from 4 to 5 components but then increases againa and is the best for 8 components on testing data but continues to increase and shows significant jump for the 8th component on the training data. MSE and RMSE similarly show a decreasing trend with increasing number of components, although not significant.

(g) ElasticNet Regularization

In [36]:
```
# elastic net regularisation

alphas = [0.01, 0.05, 0.1, 0.5, 0.9, 1]

for alpha in alphas:
    model = ElasticNet(alpha, l1_ratio=0.5, random_state=42)

    model.fit(X_train_scaled, y_train)

    y_pred = model.predict(X_test_scaled)
    y_pred_train = model.predict(X_train_scaled)

    R2 = model.score(X_test_scaled, y_test)
    R2_train = model.score(X_train_scaled, y_train)

    N = len(y_test)
    p = X_test_scaled.shape[1]

    N_train = len(y_train)
    p_train = X_train_scaled.shape[1]

    Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
    Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train

    print("alpha: ", alpha, '\n')

    print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_tr
    print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\

    print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_tra
    print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test

    print('R2 Score for train:', R2_train)
    print('R2 Score for test:', R2, '\n')

    print("Adjusted R2 Score for train:", Adjusted_R2_train)
    print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

    print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_trai
    print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n
```

```
alpha:  0.01

Mean Squared Error for train: 24478354.282398853
Mean Squared Error for test: 24248856.228290282

Root Mean Squared Error for train: 4947.560437468031
Root Mean Squared Error for test: 4924.312767106724

R2 Score for train: 0.013787910204262066
R2 Score for test: 0.0012383834956982698

Adjusted R2 Score for train: -0.0012458106767705424
Adjusted R2 Score for test: -0.06278479704944928

Mean Absolute Error for train: 4005.271158292777
Mean Absolute Error for test: 3839.381624747574

alpha:  0.05

Mean Squared Error for train: 24499905.695838828
Mean Squared Error for test: 24210445.784630973

Root Mean Squared Error for train: 4949.737942137829
Root Mean Squared Error for test: 4920.411139796244

R2 Score for train: 0.012919622073389925
R2 Score for test: 0.00282043241539387

Adjusted R2 Score for train: -0.0021273349071986303
Adjusted R2 Score for test: -0.061101334737465374

Mean Absolute Error for train: 4001.9996932398735
Mean Absolute Error for test: 3835.061865742159

alpha:  0.1

Mean Squared Error for train: 24523638.242451627
Mean Squared Error for test: 24204996.41934016

Root Mean Squared Error for train: 4952.1347157010605
Root Mean Squared Error for test: 4919.857357621271

R2 Score for train: 0.011963457940737832
R2 Score for test: 0.003044880811455708

Adjusted R2 Score for train: -0.0030980747126045927
Adjusted R2 Score for test: -0.060862498623707406

Mean Absolute Error for train: 4000.1012965943805
Mean Absolute Error for test: 3833.1716802029327

alpha:  0.5

Mean Squared Error for train: 24626260.256685033
Mean Squared Error for test: 24262212.06245762

Root Mean Squared Error for train: 4962.485290324298
Root Mean Squared Error for test: 4925.668691909518

R2 Score for train: 0.007828904206096055
```

```
R2 Score for test: 0.0006882835488326577

Adjusted R2 Score for train: -0.0072956551810061665
Adjusted R2 Score for test: -0.06337015981342153

Mean Absolute Error for train: 3997.4211264822675
Mean Absolute Error for test: 3833.3753921659923

alpha:  0.9

Mean Squared Error for train: 24668644.48692768
Mean Squared Error for test: 24291637.799691353

Root Mean Squared Error for train: 4966.753918499253
Root Mean Squared Error for test: 4928.654765723742

R2 Score for train: 0.006121279592130824
R2 Score for test: -0.0005237033840645999

Adjusted R2 Score for train: -0.009029310657989065
Adjusted R2 Score for test: -0.06465983821637633

Mean Absolute Error for train: 3998.035490384184
Mean Absolute Error for test: 3835.4456764628862

alpha:  1

Mean Squared Error for train: 24675807.87655933
Mean Squared Error for test: 24296288.079981916

Root Mean Squared Error for train: 4967.475000094045
Root Mean Squared Error for test: 4929.126502736759

R2 Score for train: 0.005832672712062448
R2 Score for test: -0.0007152390761595573

Adjusted R2 Score for train: -0.009322317033180427
Adjusted R2 Score for test: -0.06486365183745191

Mean Absolute Error for train: 3998.2727677958237
Mean Absolute Error for test: 3835.7496650742287
```

Inferences from applying ElasticNet Regularization

- As the value of alpha increases from 0.001 to 0.1 we see an increase in the R2 score but then it starts decreasing again and shows a significant drop at 1.0.
- We also see that R2 score for training data keeps on decreasing with increasing alpha values. This suggests that imposin higher penalty on the model is not a good idea as it leads to underfitting of the model.

Note: Our L1 ratio is 0.5 which means that we are using a combination of L1 and L2 regularization.

(h) Gradient Boosting Regression

In [37]:
```
# Gradient Boosting Regressor

model = GradientBoostingRegressor(n_estimators=5000, learning_rate=0.1, random_s

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)
y_pred_train = model.predict(X_train_scaled)

R2 = model.score(X_test_scaled, y_test)
R2_train = model.score(X_train_scaled, y_train)

N = len(y_test)
p = X_test_scaled.shape[1]

N_train = len(y_train)
p_train = X_train_scaled.shape[1]

Adjusted_R2 = 1 - (1 - R2) * (N - 1) / (N - p - 1)
Adjusted_R2_train = 1 - (1 - R2_train) * (N_train - 1) / (N_train - p_train - 1)

print("Iterations: ", 5000, "learning rate: ", 0.1, '\n')

print('Mean Squared Error for train:', mean_squared_error(y_train, y_pred_train)
print('Mean Squared Error for test:', mean_squared_error(y_test, y_pred), '\n')

print("Root Mean Squared Error for train:", np.sqrt(mean_squared_error(y_train,
print("Root Mean Squared Error for test:", np.sqrt(mean_squared_error(y_test, y_

print('R2 Score for train:', R2_train)
print('R2 Score for test:', R2, '\n')

print("Adjusted R2 Score for train:", Adjusted_R2_train)
print("Adjusted R2 Score for test:", Adjusted_R2, '\n')

print("Mean Absolute Error for train:", np.mean(np.abs(y_train - y_pred_train)))
print("Mean Absolute Error for test:", np.mean(np.abs(y_test - y_pred)), '\n')
```

```
Iterations:  5000 learning rate:  0.1

Mean Squared Error for train: 7.248172357591522
Mean Squared Error for test: 28777559.87192035

Root Mean Squared Error for train: 2.692242997500694
Root Mean Squared Error for test: 5364.472003088501

R2 Score for train: 0.999999707977296
R2 Score for test: -0.18528981103844022

Adjusted R2 Score for train: 0.9999997035257304
Adjusted R2 Score for test: -0.261269927130648

Mean Absolute Error for train: 2.1096410858446415
Mean Absolute Error for test: 4232.535371799777
```

Inferences from applying Gradient Boosting Regression

- On training data we see hugely significant improvement in metrics as the R2 score comes close to 1 and MSE and RMSE are very low. This suggests that the model is trained very well on the training data.
- However, on testing data we see that the R2 score is very low and MSE and RMSE are very high. This suggests that the model is overfitting on the training data and is not able to generalize well on the testing data.
- As compared to ElasticNet Regularization, Gradient Boosting Regression has performed better on the training data but has not been able to generalize well on the testing data. Hence, ElasticNet Regularization is better when it comes to generalizing on unseen data but Gradient Boosting Regression is better at training on the training data.

In [ ]: